Front. Comput. Sci., 2017, 11(6): 1061–1074
DOI 10.1007/s11704-016-5482-x

RESEARCH ARTICLE

TCP-ACC: performance and analysis of an active congestion control algorithm for heterogeneous networks

Jun ZHANG (✉)[1], Jiangtao WEN[1], Yuxing HAN[2]

1 Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China
2 TCP ENGINES, San Diego CA 92101, USA

Abstract Transmission control protocol (TCP) is a reliable transport layer protocol widely used in the Internet over decades. However, the performances of existing TCP congestion control algorithms degrade severely in modern heterogeneous networks with random packet losses, packet reordering and congestion. In this paper, we propose a novel TCP algorithm named TCP-ACC to handle all three challenges mentioned above. It integrates 1) a real-time reorder metric for calculating the probabilities of unnecessary Fast Retransmit (FRetran) and Timeouts (TO), 2) an improved RTT estimation algorithm giving more weights to packets that are sent (as opposed to received) more recently, and 3) an improved congestion control mechanism based on packet loss and reorder rate measurements. Theoretical analysis demonstrates the equilibrium throughput of TCP-ACC is much higher than traditional TCP, while maintaining good fairness with regard to other TCP algorithms in ideal network conditions. Extensive experimental results using both network emulators and real network show that the algorithm achieves significant throughput improvement in heterogeneous networks as compared with other state-of-the-art algorithms.

Keywords TCP, packet reordering, wireless networks, congestion control

1 Introduction

The main function of TCP congestion control algorithms is to dynamically adjust the *congestion window* (*cwnd*) so as to send packets into the network at an appropriate rate that more fully utilizes available bandwidth without incurring congestions.

TCP congestion control algorithms typically use two mechanisms for packet losses detection. The first is sender retransmission timeout, i.e., a packet is considered lost if the sender does not receive the corresponding acknowledgment after a certain amount of time. Alternatively, the "fast retransmit" mechanism is based on the receipt of duplicate ACKs. Under this mode, the receiver keeps track of the sequence number of the received packets and generates a duplicate acknowledgement (DUPACK) for every "out-of-order" packet. If the sender receives three DUPACKs in a row, it will retransmit the "lost" data packet immediately, without waiting for the retransmission timer to expire, and reduce the congestion window at the same time. Because of their efficiency and reliability, the above two packet loss detection mechanisms are implemented in virtually all standard TCP congestion control algorithms. However, various researches show that the performances of existing algorithms degrade severely in modern wireless and long-distance environments with a high level of packet reordering and random packet losses in addition to network congestion [1]. For heterogeneous networks, not only are the bandwidth and RTT highly variable, packet losses may also occur not because of congestion, but due to impairments in the wireless links , random collapse or advanced Ethernet NICs performing [2].

Existing research on wireless LAN and wireless cellular networks [3] shows that packet reordering and random packet

Received November 13, 2015; accepted June 28, 2016

E-mail: junzhang10@mails.tsinghua.edu.cn

loss caused by channel error, mobility and communication asymmetry strongly affect TCP performance in modern wireless networks. Usually, channel errors due to relatively high bit error rate would lead to packet or ACK losses, unnecessarily triggering *Fast Retransmission* (FRetran) or *Timeout* (TO) in state-of-the-art TCP congestion control Algorithms. Due to user mobility in wireless networks as well as temporary disconnections, out-of-order packets and random delays are much more frequent in wireless networks than in traditional networks, affecting TO counting and RTT measurements. In addition, TCP packet corrupts frequently due to congestion in wireless network because of communication asymmetry. In fact, long-distance wired networks share the problem with wireless network such as out-of-order packets. Some routing protocols such as TORA [4] maintain multiple routes and use multi-path routing to transfer data. In this situation, data packets coming from different paths may not arrive at the receiver in order, which also trigger the temporary disconnections and unnecessary *Fast Retransmission*.

An advanced TCP algorithm named TCP-ACC can detect the level of packet reordering as well as random packet losses, by combining a packet reordering measurement and congestion control so as to avoid unnecessary slowing down of the data transmission rate while avoiding introducing congestion and maintaining good fairness. In this paper, we introduce a new RTT measurement considering reorder metric, giving more weights to packets that are sent (as opposed to received) more recently. We also introduce a novel concept of out-of-order probability to quantify reorder metric, as well as to distinguish random packet loss from congestion packet loss, which significantly improve the proposed algorithm behavior in presence of heterogeneous networks. Further more, in addition to software and hardware emulators, we conduct extensive experiments on large scale "live" networks all over the world. Results in multiple cities and ISPs are shown with 4G, Wi-Fi and wired connections achieving significant throughput improvement as compared with other state-of-the-art algorithms. Theoretical analysis demonstrates the equilibrium throughput of TCP-ACC is much higher than traditional TCP, while maintaining good fairness with regard to other TCP algorithms in ideal network condition.

2 Related work

For congestion control of TCP connections over wireless and long-distance links, various algorithms have been proposed [5]. Among them, TCP Westwood [6] is not a typical AIMD (additive increase multiplicative decrease) based TCP congestion control algorithm. By counting the rate of receiving ACKs, TCP Westwood could estimate the end-to-end available bandwidth. It then sets the slow start threshold to the estimated bandwidth instead of half of the current *cwnd*, so as to avoid unnecessary reduction of the congestion window in wireless networks.

TCP Hybla in [7] was designed for satellite networks, with extra benefit of performing well over long delay wireless links as well. When the network RTT is small, TCP Hybla behaves similar to TCP Reno. With network RTT becoming large, Hybla scales *cwnd* by a fraction greater than 1, thus eleminates degradation caused by high-latency in the network.

The tax-rebate algorithm [8] was based on the AIMD method. By adapting the *cwnd* adjustment in AIMD, the Tax-rebate algorithm aims at keeping the trend of congestion window variation over time in wireless networks similar to that for wired networks. The simulation shows that the algorithm performs well when the network parameters, such as the packet loss rate and the reordering rate, are given.

Some delay-based TCP congestion control algorithms, such as TCP Vegas [9] , FAST TCP [10], and FAST-FIT [11] use the queuing delay as an indication of congestion. The queuing delay is the difference between the RTT and the propagation delay. A large queuing delay means that the probability for congestion is high. As delay-based TCP algorithms do not respond to packet lossed directly, they can avoid throughput collapse from random packet losses. The down side of these approaches is their sensitivity to RTT variations. Usually large variations of RTT in wireless network makes it hard for such algorithms to estimate the queuing delay reliably.

Some research articles [12, 13] show that, in high RTT and high packet loss rate heterogeneous networks, out-of-order packets consume more resource to buffer and play back than for ordinary wired networks. As out-of-order packets also trigger the receiver to duplicate ACKs, when the number of duplicate ACKs exceeds what is allowed by the parameter *dupthresh*, TCP will classify the subsequently transmitted non-acknowledged packet as lost and initiate FRetran.

Targeted at solving the problem caused by packet reordering, some algorithms modify switch to trace the out-of-order packets [14] , and some algorithms are implemented into TCP stack. In RRTCP [15], the constant *dupthresh* (typically 3) value in other TCP algorithms is dynamically adjusted using the reordering length stored in a reordering histogram that is updated for every reordered packet. This approach reduces the impact of unnecessary FRetrans caused by packets reordering. TCP-DCR [16] delays the initiation of Fast Retransmit for one RTT after receiving the 1st DUPACK, creating a buffer time of one RTT to determine if the duplicate ACKs are caused by reordering or loss.

3 Motivation

There are several state-of-the-art TCP algorithms aiming to improve the performance for wireless network with random packet losses. Some TCP algorithms fit congestion event very well in wired network. Also some TCP algorithms are designed for packet reordering network. TCP-FIT [17] tried to improve the performance in the wireless and high BDP (bandwidth delay product) networks together, but it did not take the influence of packet reordering into consideration.

In fact, TCP flow in heterogeneous network experiences packet reordering significantly. To illustrate the phenomena of packet reordering, we design an experiment to trace the TCP flows from an AWS server in California to clients in different cities. Each client downloads 2MB file every minute from the server for 6 hours. We use the reorder density (*RD*) of [18] to measure the out-of-orderliness of the packets.

Assuming a sequence of packets $(1, 2, \ldots, N)$ transmitted over a network, and a receive index $(1, 2, \ldots, N)$ is assigned to each packet when it arrives at the receiver. In network without packet reordering and packet loss, the arriving sequence and receive index are the same for each packets. If the receive index assigned to packet m is $m + d_m$ and $d_m \neq 0$, then the packet is out of order, and d_m is the reorder length of packet m. Packet m is 1) late iff $d_m > 0$; 2) early iff $d_m < 0$; and 3) in order if $d_m = 0$. For example in Table 1, receive index of packet 4 is 3, then $d_m = -1$ for packet 4, it indicates packet 4 arrives at client 1 packet earlier.

Defining $RD(d_m)$ as the ratio of d_m values to total number of packets sent, the histogram of *RD* are shown in Fig. 1, where the x-axis represents reorder length, with negative values corresponding to early packets and positive values designating delayed packets. It matches one's intuition that the value of *RD* decreases with higher reorder length. The total reordering packet percentages in Sydney, Moscow, Hong Kong, Cairo and Seattle are 20.33%, 27.41%, 32.63%, 43.55% and 5.24%, respectively, indicating out-of-order packets as a significant phenomena of real-life network behavior.

To design a more general TCP algorithm for heterogeneous network than the existing TCP variants such as TCP-FIT, we need to take into consideration packet reordering as well as random packet losses and congestion. It inspired us to propose TCP-ACC algorithm.

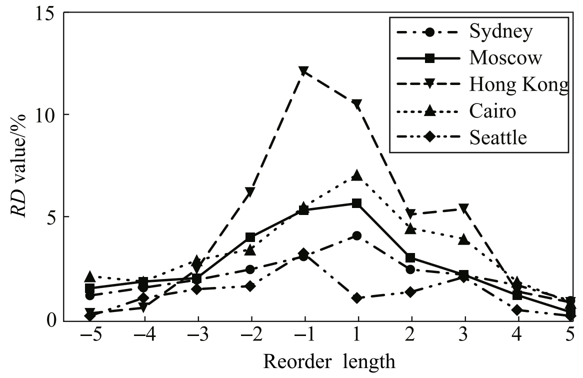

Fig. 1 Packet reordering over live network

4 TCP-ACC algorithm description

Three techniques are introduced in TCP-ACC [19] algorithm to improve TCP performance for heterogeneous networks with a high level of packet reordering. First of all, a metric to quantitatively measure reordering of packet is proposed with in-order and out-of-order packets treated separately by tracking the received ACKs. Secondly, a new RTT measurement mechanism using reorder metric is introduced to avoid unnecessary Fast Retransmit and Timeout. Thirdly, TCP-ACC uses a novel congestion control algorithm based on reorder metric and packet loss rate to improve the connection throughput. As TCP-ACC is a sender-side-only algorithm that is fully compatible with the TCP protocol, its deployment is easy and low cost.

4.1 A simplified reorder metric

The original *RD* measurement in the Linux OS requires a dynamic array to keep track of the *RD* histogram, and is therefore difficult to deploy in practical implementations of congestion control algorithms as Linux kernel modules, because the available memory is usually very limited.

In TCP-ACC, we modified the packet acknowledgement processing mechanism to find an approximate value of the reorder length and reorder density *RD* [18]. When the TCP sender receives a new ACK segment, the expected sequence number of receiver can be estimated from acknowledgement number in ACK. We denote ACK gap as the acknowledgement number displacement of two consecutive ACK segments received by TCP sender, so ACK gap should be 1 for each ACK if data segment is arriving in order. If data segment is out of order or lost, then the acknowledgement number should be the same, i.e., ACK gap should be 0, until the expected data arrives at the TCP receiver. If an ACK gap for a new arriving ACK segment is greater than 1, it indicates that TCP receiver finally received an out-of-order data segment. By examining the sequence of consecutively received ACKs, we calculate the gap between two consecutive ACKs, and then estimate the maximal reorder length and the number of reorder packets. For example, consider a sequence of packets {1, 2, 3, 4, 5, 6, 7, 8}, and assuming that the arriving sequence is {1, 2, 4, 5, 7, 8, 3, 6}, shown in Table 1. Because there are four consecutive zero values of ACK gaps, the maximal reorder length is 4. The ACK gaps for both indices 3 and 6 is 3, indicating 3 reordered packets each, namely, {3, 4, 5} reordered as {4, 5, 3} and {6, 7, 8} as {7, 8, 6}. Note that, the number of consecutive zero values for the ACK gap equals the maximal reorder length if and only if none of the packets is corrupted.

Table 1 Illustration of reorder measurement

Arriving sequence	1	2	4	5	7	8	3	6
Receive index	1	2	3	4	5	6	7	8
Reorder length	0	0	–1	–1	–2	–2	4	2
ACK	2	3	3	3	3	3	6	9
ACK gap		1	0	0	0	0	3	3

A simple way to characterize reordering in a received sequence is to count reorder packets and maximal reorder length. As packet reordering introduces duplicate acknowledgements, for networks with a high level of packet reordering, a sender should receive more duplicate ACKs than the standard setting of the *dupthresh* parameter before initiating FRetran. In the Linux OS, the value *dupthresh* is usually set to 3. If a data segment is reordered for more than three packets, the receiver will send more than three duplicated ACKs while the segment is only late. Thus we can estimate the rate of Fast Retransmit triggered by packet reordering, i.e., the probability of a segment is considered lost because of packet reordering.

$$P_o = \sum_{i \geqslant dupthresh} MRL(i)/PacketNum, \tag{1}$$

where $MRL(i)$ is the number of reordered packets whose Max Reorder Length is i, and $PacketNum$ is the total number of packets. $\sum_{i \geqslant dupthresh} MRL(i)$ is the number of Fast Retransmit events triggered by "late" segment.

We define $NumAckGap(i)$ as the number of ACK segment whose ACK gap is i. By counting the number of ACK segment whose ACK gap is greater than 1, we can get the number of out-of-order data segment. Thus we can estimate the rate of out of ordering, i.e., the probability of a segment could not arrive in order.

$$P_{ooo} = \sum_{i>1} NumAckGap(i)/PacketNum. \tag{2}$$

Packets with $ACKGap = 1$ are excluded in P_{ooo} calculation as they are in order packets.

4.2 RTT measurement considering reorder metric

As packets sent and ACKs received, Linux OS keeps an exponentially weighted moving average of smoothed RTT values. The estimate used named as $SRTT$ is given by:

$$SRTT_{new} = (1-\alpha) \times SRTT_{old} + \alpha \times rtt\_sample, \tag{3}$$

where $0 \leqslant \alpha < 1$ is a constant, and *rtt_sample* is the RTT value of the packet whose acknowledgement has just arrived. From the equation, it can be seen that any recent variation in *rtt_sample* has a significant impact on the final estimation. Such a mechanism works well for network conditions whose acknowledgments arrive in order. However, for networks with a high level of reordering packets, it is appropriate to give more weight to packets that are sent (as opposed to have arrived) most recently, as they reflect the latest state of the network.

Therefore, an improved RTT estimate $RSRTT$ is defined as

$$RSRTT_{new} = (1-\alpha) \times RSRTT_{old} + \alpha \times rtt_{last}, \tag{4}$$

where rtt_{last} is an exponentially weighted average of past RTT values that is updated whenever a new ACK arrives using

$$rtt_{last} = \max\left\{rtt_{last}, rtt\_sample/\beta^{ReorderLength}\right\}, \tag{5}$$

where $0 \leqslant \beta < 1$ is a constant, *ReorderLength* is 0 when $AckGap > 0$, otherwise *ReorderLength* is the number of continuous zero-valued $ACKGap$. Note that in the calculation of rtt_{last}, when the reordering length of current ACK is large, the current RTT observation would carry less weight. Whereas when there is no out-of-order packet, Eq. (4) would be identical to Eq. (3). In this paper, the parameter α is set to $1/8$ as in Eq. (3), and β is set to $15/16$, thus *rtt_sample* can change slightly for each duplicated ACK.

Inside TCP stack, Timeout Phase was triggered when the ACK of a packet is not received in the RTO (retransmission timeout) time period. Our proposed modified RTT measurement contributes to keeping RTO estimation to a more appropriate level by considering packet reordering. In Subsection 6.1, experimental results show this new RTT and RTO estimation technique effectively compensates for random RTT spikes of packet reordering due to network mobility or asymmetry.

4.3 Compensated congestion control (CC) for TCP-ACC

The traditional Additive-Increase-Multiplicative-Decrease (AIMD) *cwnd* update mechanism used in many TCP algorithms works as follows:

$$\begin{aligned}
&\text{Each ACK}:\\
&\quad cwnd = cwnd + 1, cwnd < ssthresh;\\
&\quad cwnd = cwnd + \tfrac{1}{cwnd}, cwnd \geqslant ssthresh.\\
&\text{Each loss}:\\
&\quad ssthresh = cwnd/2,\\
&\quad cwnd = cwnd/2.
\end{aligned} \tag{6}$$

In Eq. (6), the value *ssthresh*, known as the slow start threshold, is critical for determining when to increase the congestion window rapidly. In wireless networks where fast transmissions and timeouts might be caused by congestion, random wireless losses or packet reordering, the congestion window calculated using the standard AIMD algorithm tends not to fully utilize the network bandwidth.

In heterogeneous network with packet reordering, fast transmissions and packet losses may be caused by DUPACKs, congestion or random packet losses. In TCP-ACC, we use the reorder metric and packet loss measurement to avoid unnecessary FRetran and throughput collapse. We estimate the congestion probability by counting the number of FRetran and TO function calls. Obviously, larger random packet loss rate and out-of-order probability could trigger more FRetran and TO functions calls. The packet loss probability is estimated as:

$$P_{loss} = [num(FRetran) + num(TO)] / PacketsNum, \tag{7}$$

where $num(FRetran)$ and $num(TO)$ are the numbers of the two function calls respectively.

As defined in Eq. (7), the probability for a packet to be considered lost is P_{loss}, which is the sum of the probabilities for the packet to be lost due to congestion, random wireless loss, as well as reorder-introduced retransmission. Thus we have:

$$P_{loss} = P_c + P_r + P_o, \tag{8}$$

where P_c, P_r and P_o denote the probabilities for a packet to be considered lost due to congestion, random error and out-of-order retransmission, respectively.

As it is difficult for the sender to distinguish in real time between a packet lost due to congestion and a packet lost due to random errors, we assume that a packet is lost due to congestion only when the queuing delay of previously ACKed packet was larger than $\zeta \times q_{\max}$, where $\zeta = \frac{1+P_{ooo}}{2}$ is decided by P_{ooo} and $q_{\max}$ is the maximum queuing delay. We can then count the numbers of congestion events and random loss events accordingly.

The probability for a packet to be considered lost due to reasons other than congestion is P_f/P_{loss}, where $P_f = P_r+P_o$ is the non-congestion loss probability, in which case the values of *cwnd* and *ssthresh* should not be decreased. Otherwise, *cwnd* and *ssthresh* could use the traditional adjustment equations. On average therefore,

$$\begin{aligned}
&\text{Each loss}:\\
&ssthresh = \frac{P_f}{P_{loss}} \times cwnd + (1 - \frac{P_f}{P_{loss}}) \times \frac{1}{2} \times cwnd,\\
&\qquad = \frac{P_f + P_{loss}}{2P_{loss}} \times cwnd,\\
&cwnd = ssthresh.
\end{aligned} \tag{9}$$

Notice that P_f is the probability of non-congestion packet loss, it should be 0 in ideal network. Thus, Eq. (9) would be the same as the AIMD algorithm (Eq. (6)) in ideal network.

After the fast retransmit following a packet loss, TCP-ACC enters the congestion avoidance phase. For a TCP session using the AIMD congestion window adjustment in well-behaved networks with no random losses and no packets reordering, the congestion window will grow from $\underline{W}$ to $\overline{W}$, until the session experiences a congestion-caused packet loss.

As congestion window in AIMD adjustment method is increased by 1 for each RTT, then it will take $\overline{W} - \underline{W}$ RTT interval to update the congestion window. The average congestion window in these RTTs is $(\overline{W} + \underline{W})/2$. Thus in this period, a total of $(\overline{W}^2 - \underline{W}^2)/2$ packets were sent, with one congestion loss, we can estimate the congestion loss probability.

$$P_c = \frac{2}{\overline{W}^2 - \underline{W}^2}. \tag{10}$$

The same holds for heterogeneous networks, if P_c is replaced by P_{loss}

$$P_{loss} = \frac{2}{\overline{W}^2 - \underline{W}^2}, \tag{11}$$

which means,

$$\overline{W} = \sqrt{\underline{W}^2 + \frac{2}{p_{loss}}}. \tag{12}$$

Accordingly, we modify Eq. (12) and use the following to set the appropriate congestion window:

$$\text{Each ACK}: \\ cwnd = cwnd + \min\left\{\frac{\sqrt{cwnd^2 + \frac{2}{p_{loss}}}}{cwnd} - 1, \frac{\delta}{cwnd}\right\}. \tag{13}$$

For the sake of simplicity, in our implementation of the algorithm as a kernel module in the Linux operating system, we used the Newton-Raphson method to approximate the square root, and prevented the increment of *cwnd* in single round from exceeding a threshold δ. The increment of congestion window for each ACK is greater than 0 and less than $\delta/cwnd$, by choosing a proper value of δ, we can have an average increment around $1/cwnd$, which is the increment value in AIMD algorithm (Eq. (6)). As we should estimate packet loss rate (P_{loss}) at first to change the congestion window.

If the TCP flow is too short to calculate the P_{loss}, i.e., there is no packet loss in the flow, then the performance of TCP-ACC will be the same as that of TCP-Reno, which will be less than some fast startup TCP variants such as TCP Hybla and TCP FIT. For high BDP (bandwidth delay product) network with low packet loss rate and packet reordering rate, it will take a lot of RTTs to increase the Congestion Window to a proper situation after each packet loss for TCP-ACC and other AIMD-like TCP variants, the throughput performance is usually worse than delay-based TCP such as Fast TCP. It is the tradeoff to keep TCP-ACC fair with other traditional TCP such as TCP Reno in these network environment.

To reduce the impact of short-term network variations, we update the probabilities, P_o, P_{ooo}, P_r and P_{loss} using a fixed time interval, which was set to one second in the experiments. At the end of the current time period, we update the probabilities using Kalman filter:

$$P^{nxt} = \frac{7}{8} \times P^{pre} + \frac{1}{8} \times P^{sample}, \tag{14}$$

while P^{pre} is the corresponding probability of the previous time period, and P^{sample} is the probability calculated for the current time period according to Eq. (7).

5 Throughput model of TCP-ACC

In this section, we use the approximate stable state throughput model to analyze the performance of TCP-ACC in wireless networks. The model was originally introduced in [20], and was also used in [17,21].

Considering the periods between two continuous packet losses indicated by three duplicate ACKs, we denote the congestion windows size at the end of ith period as W_i, and the duration of ith period as A_i. Without loss of generality, we assume that there are X_i rounds of increments to the value of the congestion window. For each round, congestion windows is increased to

$$\sqrt{W_{cur}^2 + \frac{2}{p_{loss}}}, \tag{14}$$

according to Eq. (13). For each packet loss, congestion window is decreased to $\gamma \times W_{cur}$, where

$$\gamma = \frac{P_f + P_{loss}}{2P_{loss}}, \tag{15}$$

according to Eq. (9). For simplicity of notations, we will denote p_{loss} as p in the subsequent analysis.

For the ith period, we assume that the number of packets sent is Y_i. Then the expected equilibrium throughput should be

$$T = \frac{E[Y]}{E[A]}. \tag{16}$$

Assuming as in [20] that $Y_i = a_i + W_i - 1$, where a_i is the number of first packet lost in ith period, then obviously, the expected value of a is

$$E[a] = \sum_{k=1}^{\infty} (1-p)^{k-1} pk = \frac{1}{p}. \tag{17}$$

Thus we have,

$$E[Y] = \frac{1}{p} + E[W] - 1. \tag{18}$$

At the beginning of the ith period, the value of congestion window is $\gamma \dot{W}_{i-1}$. After the first round without packet loss, congestion window would increase to $\sqrt{(\gamma W_{i-1})^2 + \frac{2}{p}}$, or $\sqrt{(\gamma W_{i-1})^2 + \frac{2k}{p}}$ after the kth round. Because there are X_i rounds in the ith period,

$$\begin{aligned} Y_i &= \sum_{k=0}^{X_i} \sqrt{(\gamma W_{i-1})^2 + \frac{2k}{p}} + b_i \\ &\simeq \frac{p}{3}((\gamma W_{i-1})^2 + \frac{2X_i}{p})^{\frac{3}{2}} - \frac{p}{3}(\gamma W_{i-1})^3 + b_i, \end{aligned} \tag{19}$$

where b_i is the number of packets sent in the last round, and

$$W_i = \sqrt{(\gamma W_{i-1})^2 + \frac{2X_i}{p}}. \tag{20}$$

From Eq. (20), we have

$$E[X] = \frac{p}{2}(1-\gamma^2)E[W]. \tag{21}$$

As b_i is a random value smaller than W_i, we have $E[b] = E[W]/2$. Combining Eqs. (19) and (21), we can find the expected value of Y_i:

$$E[Y] = \frac{p}{3}(1-\gamma^3)E[W]^3 + \frac{1}{2}E[W]. \tag{22}$$

The expected value of W_i can be found from Eqs. (22) and (18),

$$\frac{1}{p} + E[W] - 1 = \frac{p}{3}(1-\gamma^3)E[W]^3 + \frac{1}{2}E[W],$$
$$E[W] \approx (\frac{3(1-p)}{p^2(1-\gamma^3)})^{\frac{1}{3}}.$$

Denote the average RTT in the ith period as D_i, then the expected value of A_i is

$$\begin{aligned} E[A] &= E[X]E[D] \\ &= \frac{p}{2}(1-\gamma^2)E[W]E[D] \\ &= \frac{p}{2}(1-\gamma^2)(\frac{3(1-p)}{p^2(1-\gamma^3)})^{\frac{1}{3}}E[D]. \end{aligned} \tag{23}$$

From Eqs. (23), (22) and (16), the equilibrium throughput

$$\begin{aligned} T &= \frac{\frac{1-p}{p} + E[W]}{E[A]} \\ &= \frac{\frac{1-p}{p} + (\frac{3(1-p)}{p^2(1-\gamma^3)})^{\frac{1}{3}}}{\frac{p}{2}(1-\gamma^2)(\frac{3(1-p)}{p^2(1-\gamma^3)})^{\frac{1}{3}}E[D]}. \end{aligned} \tag{24}$$

When p is sufficiently small, the equilibrium throughput can be simplified to

$$T = \frac{(1-\gamma^3)^{\frac{1}{3}}}{p^{\frac{4}{3}}(1-\gamma^2)E[D]}. \tag{25}$$

According to Eq. (15), we have $1/2 \leqslant \gamma < 1$. As $(1-\gamma^3)^{\frac{1}{3}}/1-\gamma^2$ is an increasing function of γ, then we have,

$$T \geqslant \frac{1.28}{p^{\frac{4}{3}}E[D]}. \tag{26}$$

Compared with TCP Reno's equilibrium throughput in [20], $T_{reno} = \sqrt{\frac{3}{2}}/p^{\frac{1}{2}}E[D]$, the equilibrium throughput of the TCP-ACC algorithm is much higher as $0 < p < 1$.

Using the modified growth function defined in Eq. (13) when p is very small, e.g., for ideal wired networks, the behavior of the congestion window update for the TCP-ACC algorithm is similar to the origin AIMD algorithm. This characteristic ensures that the TCP-ACC algorithm maintains good fairness with regard to other TCP algorithms in wired networks.

6 Experimental evaluations

We evaluat the performance of the proposed TCP-ACC algorithm using software emulator (Network Simulator 2 (NS-2)), hardware emulator (Linktropy), and “live” networks in multiple cities and ISPs all over the world.

6.1 Simulation result

Extensive simulations are conducted on NS-2. NS-2 is a reknowned network event simulator supported varies TCP, routing and multicast protocols. In simulation, TCP Westwood, TCP Hybla, TCP SACK [22], RRTCP and TCP-DCR algorithms are used for comparison. δ in Eq. (13) is set to 5.

The simulation topology is illustrated in Fig. 2, in which different TCP senders $\{S1, S2, \ldots\}$ are connected to different receivers $\{D1, D2, \ldots\}$ via a bottleneck $(R1, R2)$. The connections from the senders and $R1$ and $R2$ to the receivers are error-free wired links with a bandwidth of 10Mbps and 1ms of delay. The connection between $R1$ and $R2$ is a simulation link, with the bandwidth and baseRTT set to 10Mbps and 50ms. The random packet loss rate for the bottleneck link $(R1, R2)$ varies from 0% to 5%, while the packet reordering rate varied from 0% to 30%. In the experiments, the delays of the reordered packets follow a normal distribution with the mean and standard deviation set to 25ms and 8ms respectively.

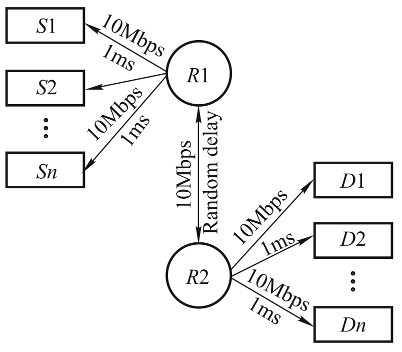

Fig. 2 Simulation topology

First, performance of the proposed improved RTT measurement algorithm as described in Subsection 4.2 is studied. In this implementation, we set $\alpha = 1/8$ and $\beta = 15/16$. Figure 3 shows the throughput of TCP CUBIC with traditional RTT measurment vs. improved RTT mesurement, where x-axis is packet reorder rate and y-axis is network throughput. On average, 37% and 23% gains are achieved using the proposed algorithm with network packet loss rate (PLR) at 0% and 1% respectively. Besides, this further justifies how packet reordering could impact network performance. For example, in the case where packet reorder rate is as small as 0.01, with 0% PLR the throughput of the traditional algorithm is 1,610Kbps while the proposed algorithm is 1,920Kbps, i.e., at least 19% performance loss because of packet reordering. It is also verified in the experiemnt in the case of no packet reordering, both algorithms could fully utilize bandwidth and arrive at the same throughput number indicating Eq. (4) becomes identical as Eq. (3).

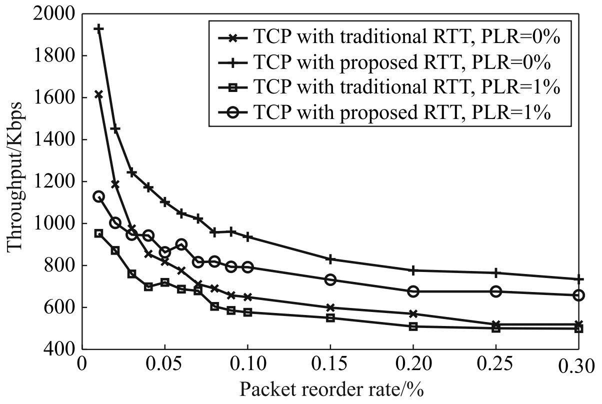

Fig. 3 Throughput comparison of traditional and proposed RTT mesurement

Next, the complete TCP-ACC congestion control algorithm is implemented and studied. Throughput achieved by RRTCP, TCP Reno (with SACK), TCP DCR, TCP Westwood, TCP Hybla and TCP-ACC algorithms are plotted as a function of packet reorder rate in Fig. 4. In the case of no packet loss rate in Fig. 4(a), TCP-ACC, TCP DCR and RRTCP algorithms achieve significant throughput improvements over other TCP algorithms that have not been designed to handle packet reordering. Figure 4(b) shows the results for packet reordering networks with a random packet loss rate of 1%. The TCP-ACC algorithm still achieves significant throughput gain with a bandwidth utilization greater than 70%, whereas the performances of the TCP DCR and RRTCP algorithms severely degrade to about 17% and 12% bandwidth utilization. Similar observations can be made from Fig. 4(c).

To examine the fairness of the algorithm, ten senders are used to transmit packets to their corresponding receivers simultaneously, and the system wide inter-flow fairness are calculated using Jain's Fairness Index (JFI) [23],

$$J = \frac{(\sum_{i=1}^{n} r_i)^2}{n\sum_{i=1}^{n} r_i^2}, \qquad (27)$$

where n is the number of flows, r_i is the throughput of flow i. In Eq. (27), $J = 1$ indicates complete fairness across all flows, while $J = 0$ means there is no fairness at all. In general, a TCP algorithm is considered as "fair" with $J > 90\%$. According to Eq. (27), JFI is independent of the throughputs of the flows, and thus is applicable across groups of TCP flows with different aggregate throughputs.

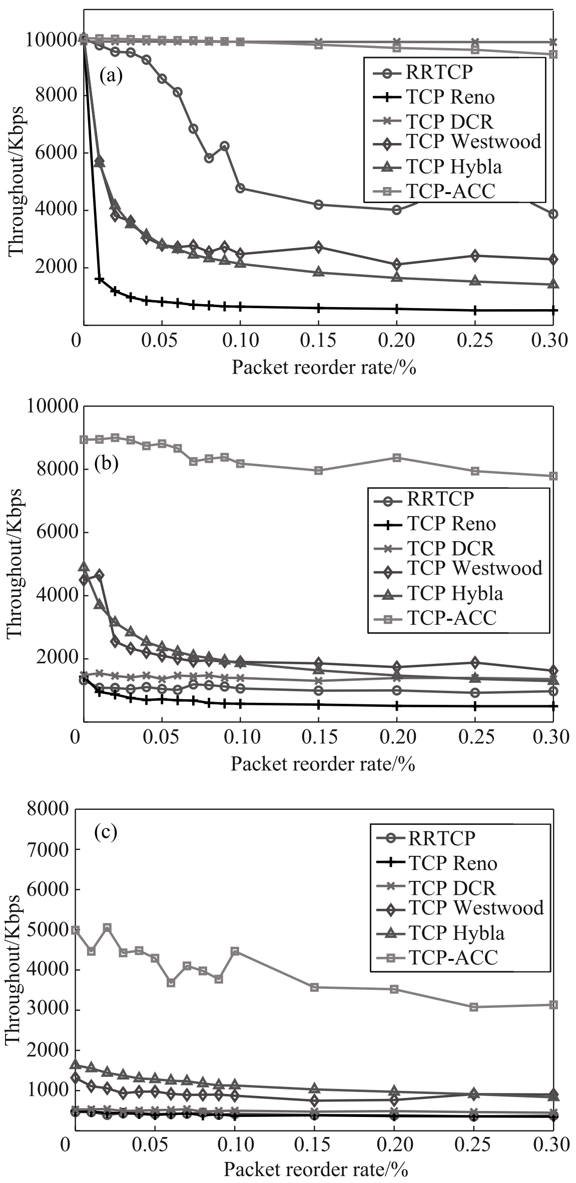

Fig. 4 Throughput vs. packet reordering. (a) PLR = 0%; (b) PLR = 1%; (c) PLR = 5%

JFI values of TCP-ACC as a function of packet loss rate are plotted in Fig. 5, with packet reorder rate varying from 0% to 30%. The random packet loss rate of bottleneck ($R1, R2$) is set at 1% and 5%. It can be observed regardless of various network conditions, TCP-ACC achieves good JFI fairness above 90%, indicating the algorithm has good intra-fairness.

6.2 Testbed experiments

In addition to simulations using the NS-2, we connect a Linux server with the TCP-ACC algorithm to PC clients via a hardware wireless network emulator named Linktropy. Hardware emulator like Linktropy has extra benifit of setting network parameters without affecting TCP stack on sender. The simulation topology is illustrated in Fig. 6.

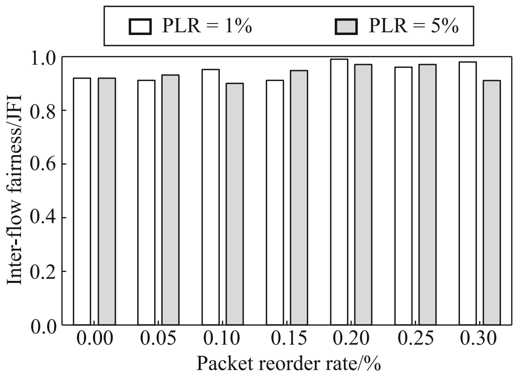

Fig. 5 Inter-flow fairness of TCP-ACC

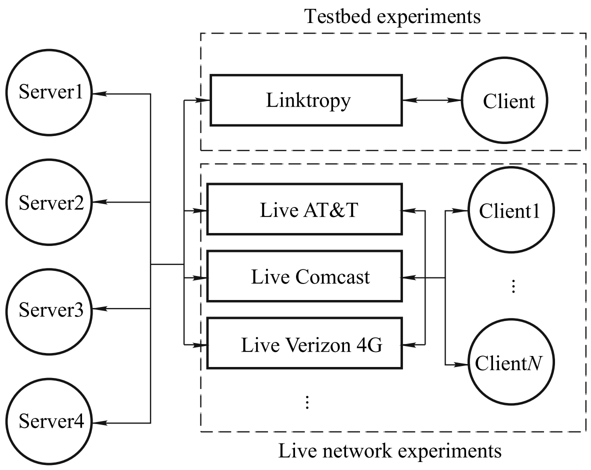

Fig. 6 Testbed and live experiments topology

We implement TCP-ACC algorithm in Linux servers. In order to update the probabilities, P_o, P_r, and P_{loss} every second, we have to keep a timer in Linux implementation to trigger the updating method. What's more, modifications in RTT measurement function, fast retransmission function and timeout function are needed to evaluate packet reordering and packet losses for TCP-ACC, the same modification is unnecessary for other TCP variants in Linux Kernel. We use 64 bytes in kernel module to trace the state of TCP parameters for each TCP-ACC flow, while other TCP algorithms usually cost 30–40 bytes for each flow. Thus the memory occupation of TCP-ACC is slightly larger than that of other algorithms.

We first use the "tcpprobe" kernel module to trace the values of *cwnd* and *ssthresh* for the TCP-ACC algorithm in the Linux kernel. The random packet loss rate is set to 0.001% and propagation delay is set to 50ms. As shown in Fig. 7(a), the value of *cwnd* keeps increasing rapidly when *cwnd* is small or when a random packet loss (i.e., not due to congestion) occurs, while the growth of *cwnd* slows down when its value is large enough. Figure 7(b) shows the *cwnd* and *ssthresh* of TCP Reno, which uses the origin AIMD method, under the same network conditions. As the Additive Increase ("AI") phase for the Reno algorithm is not able to "probe" the network condition fast enough for the value of *cwnd* to reach a reasonable value before a random packet loss occurs, the average throughput of TCP Reno is only half of that for TCP-ACC.

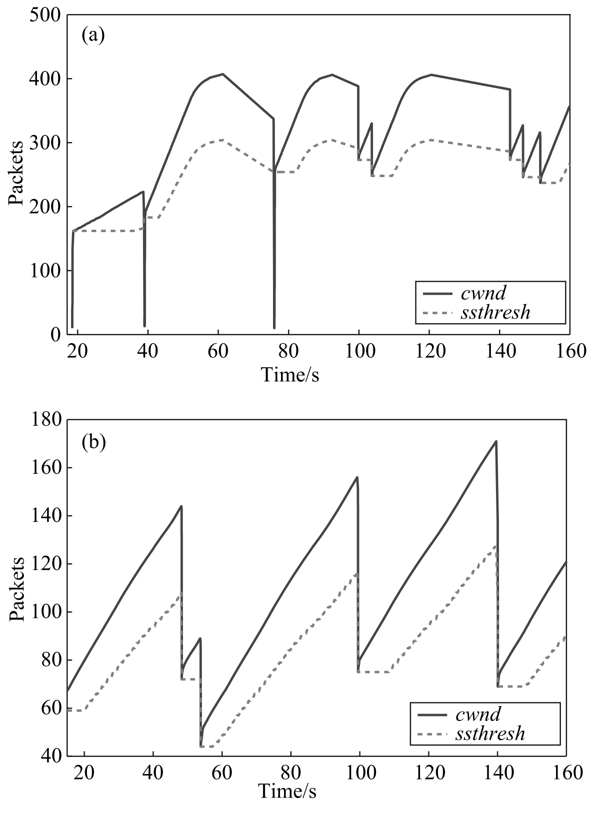

Fig. 7 Trace of *cwnd* & *ssthresh*. (a) TCP-ACC; (b) TCP Reno

Next, we test the performance of the proposed algorithm on a relatively stable network such as LAN or Backbone connections in a CDN or data center with relatively stable RTT and small PLR. The parameter of bandwidth is set at 100Mbps with the random packet loss varying from 0.001% to 1%, and propagation delay set at multiple constant values. In the experiment, throughput of TCP-ACC, Reno, Cubic, Westwood and Hybla are ploted in Fig. 8. Each test lasts for 300 seconds and is then repeated 20 times. It is observed that TCP-ACC achieves higher throughput than all other TCP algorithms. In the case where random packet loss rate is relatively high, TCP-ACC achieves as high as 100%+ speedup as compared to other TCP variants.

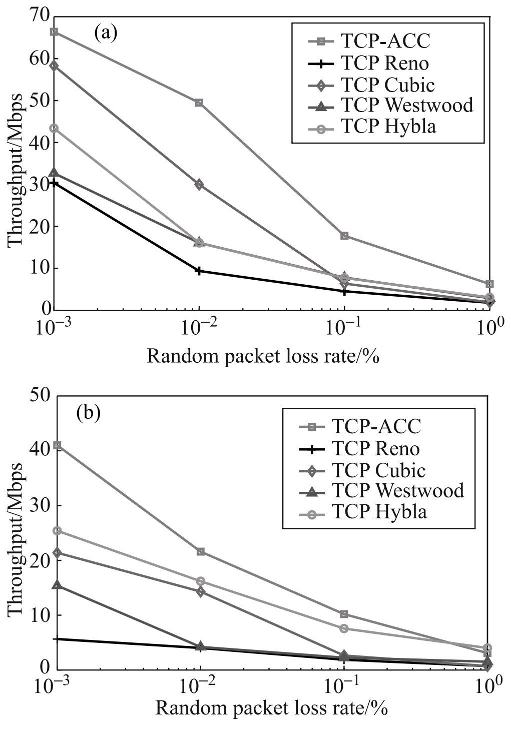

Fig. 8 Throughput vs. random packet loss rate with constant RTT for long-duration flows. (a) Constant RTT is 50ms; (b) constant RTT is 150ms

In addition, we test performance of the proposed algorithm in a varying network condition such as WLAN or Lastmile connections in a CDN or data center with varying RTT and relatively high PLR. The parameter of bandwidth is set at 30Mbps with random packet loss rate varying from 0.01% to 5%, and propagation delay set as 50ms and 150ms with standard deviation 16ms and 50ms following Normal distribution. As illustrated in Fig. 9, TCP-ACC maintains significant performance gain over other TCP variants.

Then we test the performance of short-duration flows. The parameter of bandwidth is set at 100 Mbps with random packet loss rate varying from 0.001% to 5%, and propagation delay set as 50ms with or without standard deviation 16ms. Each test lasts for 2 seconds and is repeated for 100 times. We do not set the propagation delay to 150ms this time, because 150ms is too long for a 2-second duration flow. As illustrated in Fig. 10, TCP-ACC still performs better than other TCP variants on this kind of short-duration flows.

To test TCP-friendliness, we keep two servers sending data competing in the same network whose bandwidth is 100Mbps and RTT is 25ms, one is using TCP Reno, and the other is using TCP Reno, TCP-ACC, Cubic, Westwood and Hybla. We keep each algorithm transmitting for 200 seconds and record the average throughput separately. First, we test the ideal network environment whose PLR is 0. The result in Fig. 11(a) shows TCP-ACC and TCP Cubic are most TCP-friendly in ideal network. Second, we record the throughput of two servers in network whose PLR is 1%. The result in Fig. 11(b) shows that the throughput of TCP-ACC and Cubic is quite different from the throughput of others. It is because TCP-ACC and TCP Cubic could utilize the bandwidth that the others cannot use in high PLR network.

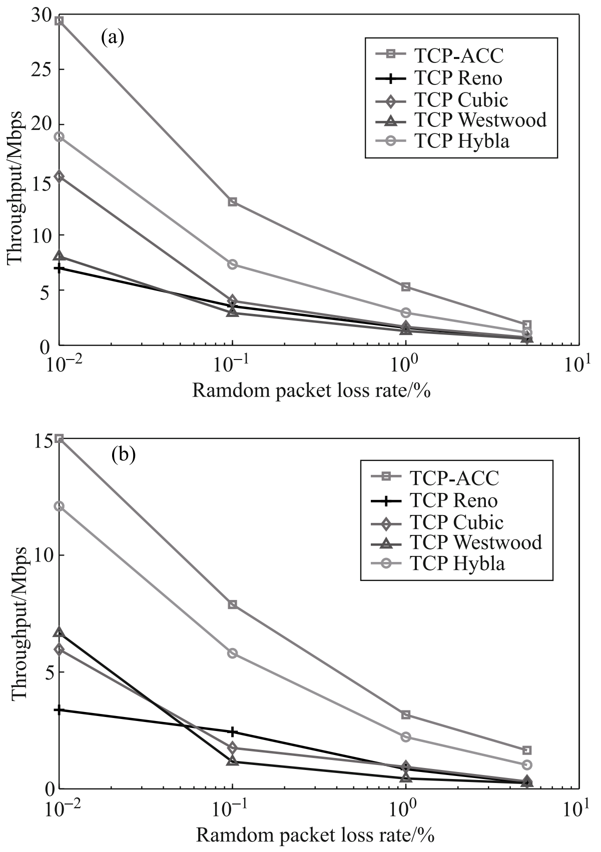

Fig. 9 Throughput vs. random packet loss rate with RTT followed Normal distribution for long-duration flows. (a) Mean RTT is 50ms and standard deviation is 16ms; (b) mean RTT is 150ms and standard deviation is 50ms

6.3 Live network experiments

Besides software simulation of NS-2 and hardware testbed of Linktropy, extensive experiments are conducted in live networks. The experiment topology is similar to Fig. 6, except that the connection links between server and clients are live wireless (4G, 3G and WiFi) and wired (Backbone and Lastmile) networks. Four servers are set up at the same location in Southern California with each server impelmenting different TCP algorithm, i.e., TCP Cubic, TCP Reno, TCP Westwood and TCP-ACC. 235 global nodes in six continents with wired, WiFi, 3G and 4G connections are selected to pull the same files from these servers. The transmitted file size is set to 2MB in order for algorithms tested to reach steady-state. In addition, this file size is consistent with slice size of ordinary http streaming video such as YouTube, Yahoo and Netflix.

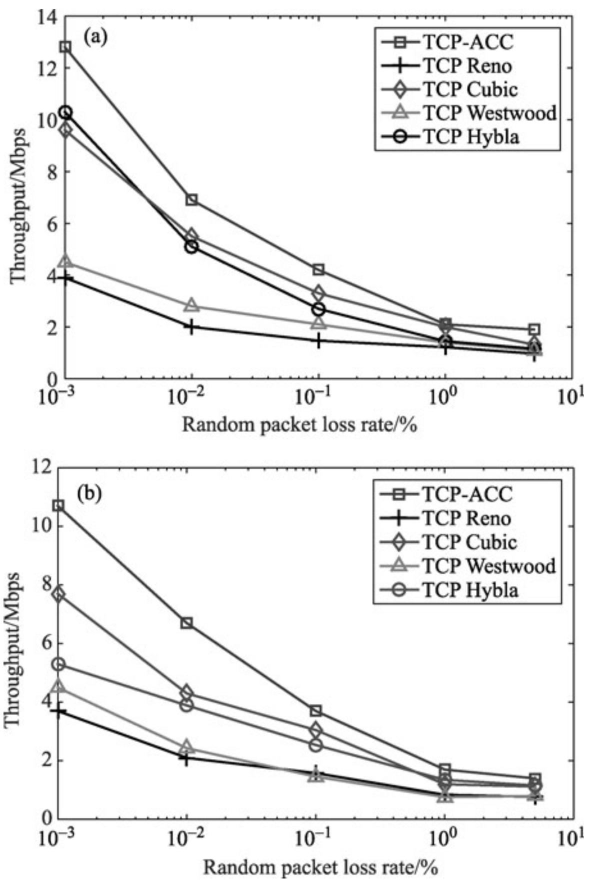

Fig. 10 Throughput vs. random packet loss rate for short-duration flows. (a) Mean RTT is 50ms; (b) mean RTT is 50ms, standard deviation is 16ms

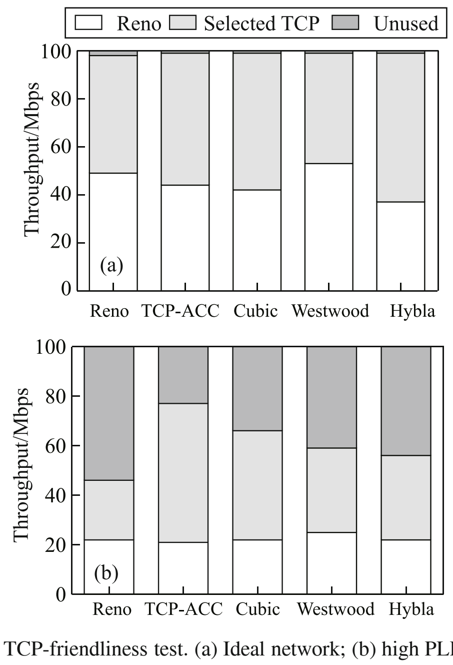

Fig. 11 TCP-friendliness test. (a) Ideal network; (b) high PLR network

Figures 12 and 13 illustrate throughput of the above four algorithms over live network observing by catchpoint, which is a World Wide Web performance monitoring system, in different locations and ISPs. In Fig. 12, TCP-ACC shows superior performance over other three algorithms in multiple cities. For each node, 25 tests are conducted with average throughput calculated. In Fig. 13, multiple ISPs are selected to illustrate algorithm performance over different network conditions, where AT&T 4G and Verizon 4G representing 4G connections, AT&T uverse, Comcast and Google Filber representing Lastmile connections, and Level3, NTT Verizon and Zayo representing Backbone connections. It is observed that TCP-ACC achieves the highest throughput among all TCP variants by significant amount.

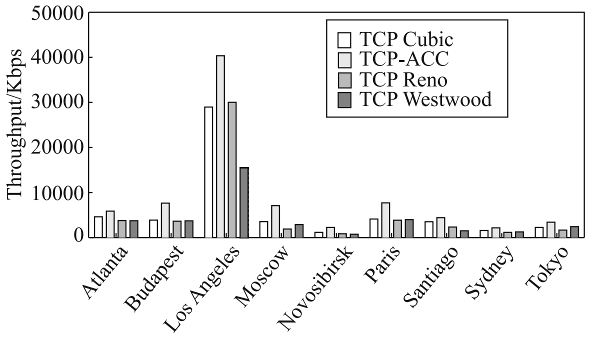

Fig. 12 Throughput of different locations over live network

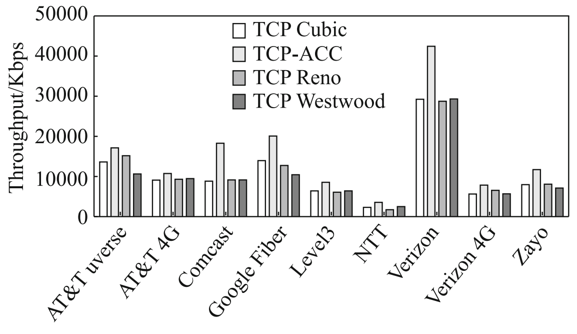

Fig. 13 Throughput of different ISPs over live network

Figure 14 shows the load time of 2MB file in multiple cities over a certain time period. In general, TCP-ACC remains the fastest speed. For example, in the city of Paris, TCP-ACC performs almost consistantly twice as fast as other algorithms. In cities where big "spikes" occur in other algorithms which is possibly caused by poor network condition and thus number of out-of-order packets is high (i.e., Atlanta, Moscow and Santiago, etc.), variantion in TCP-ACC is a lot smoother.

We test performance of long-duration flows over 3G-wired hybrid network, the server in Beijing is connected to the

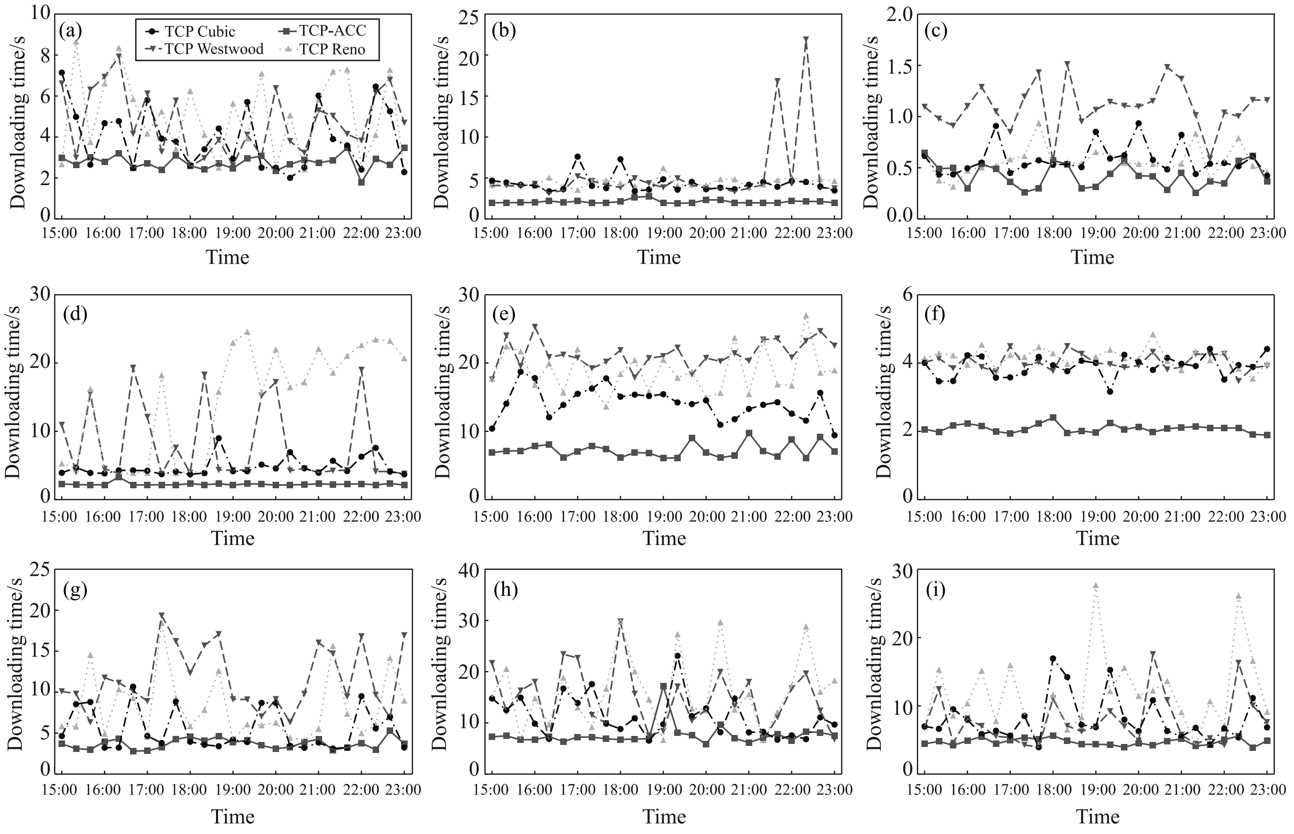

Fig. 14 Load time of 2MB file over live network. (a) Atlanta; (b) Budapest; (c) Los Angeles; (d) Moscow; (e) Novosibirsk; (f) Paris; (g) Santiago; (h) Sydney; (i) Tokyo

Internet via a 3G USB key, four clients located in Beijing, Singapore, Los Angeles and Paris download 500MB file from the server. The TCP congestion control algorithms on the server are TCP-ACC, TCP Reno, TCP Cubic and TCP Westwood. Figure 15 shows the throughput collected by four clients. The result shows that TCP-ACC always works better than the TCP Reno and TCP Westwood, TCP Cubic is slightly better only in Los Angeles case, and overall TCP-ACC works well for long-duration flows in live network.

The intra-protocol fairness of TCP-ACC algorithm for parallel connections is investigated in Fig. 16. We set the server and client on a LAN of 100Mbps bandwidth and the 20ms RTT. A series of five TCP-ACC flows were started at 20-second intervals, each lasting 100 seconds. The throughputs of the five TCP-ACC connections are shown in Fig. 16. The combined throughput of multiple connections is close to the link capacity of 100 Mbps, with the throughput of each flow stabilizing quickly at their fair share.

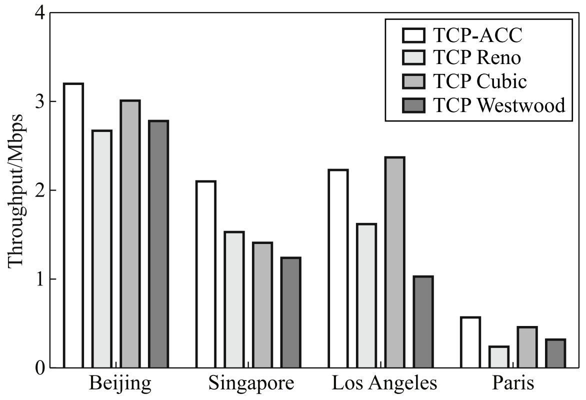

Fig. 15 Long-duration flows over live 3G network

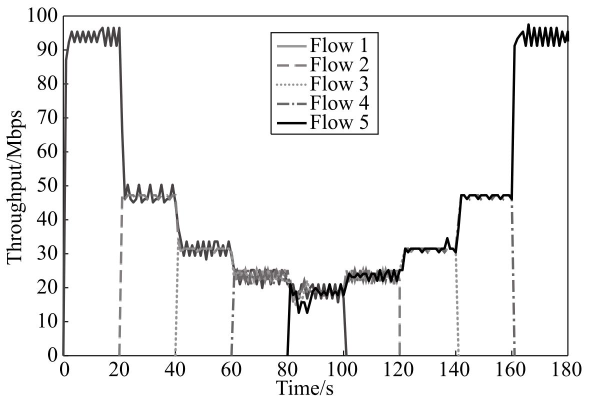

Fig. 16 Intra-protocol fairness of TCP-ACC over live network

7 Conclusion

In this paper, we describe a novel TCP algorithm, namely TCP-ACC that achieves good throughput and inter-flow fairness in the presence of congestion, random packet losses and packet reordering. It integrates 1) a real-time reorder metric for calculating the probabilities of unnecessary FRetrans and TOs, 2) an improved RTT estimation algorithm giving more weights to packets that are sent (as opposed to received) more recently, and 3) an improved congestion control mechanism based on packet loss and reorder rate measurements. Simulations and "live" network test show that the performance of the algorithm is significantly improved over other TCP algorithms designed for networks with packet reordering and random packet losses while achieving very good intra-algorithm fairness and TCP-friendliness.

Acknowledgements This work was supported by the National Science Fund for Distinguished Young Scholars of China (61125102) and the State Key Program of National Natural Science Foundation of China (Grant No. 61133008).

References

1. Leung K C, Li V O K, Yang D Q. An overview of packet reordering in transmission control protocol (TCP): problems, solutions, and challenges. IEEE Transactions on Parallel and Distributed Systems, 2007, 18(4): 522–535
2. Wu W J, DeMar P, Crawford M. Why can some advanced ethernet nics cause packet reordering? IEEE Communications Letters, 2011, 15(2): 253–255
3. Chen X, Zhai H Q, Wang J F, Fang Y G. A survey on improving TCP performance over wireless networks. In: Cardei M, Cardei I, Du D Z, eds. Resource Management in Wireless Networking. Network Theory and Applications, Vol 16. Springer US, 2005, 657–695
4. Park V D, Corson M S. A highly adaptive distributed routing algorithm for mobile wireless networks. In: Proceedings of the 16th Annual Joint Conference of the IEEE Computer and Communications Societies. 1997, 1405–1413
5. Afanasyev A, Tilley N, Reiher P, Kleinrock L. Host-to-host congestion control for TCP. IEEE Communications Surveys & Tutorials, 2010, 12(3): 304–342
6. Mascolo S, Casetti C, Gerla M, Sanadidi M Y, Wang R. TCP westwood: bandwidth estimation for enhanced transport over wireless links. In: Proceedings of the 7th International Conference on Mobile Computing and Networking. 2001, 287–297
7. Caini C, Firrincieli R. TCP hybla: a TCP enhancement for heterogeneous networks. International Journal of Satellite Communications and Networking, 2004, 22(5): 547–566
8. Lai C D, Leung K C, Li V O. Design and analysis of TCP aimd in wireless networks. In: Proceedings of IEEE Wireless Communications and Networking Conference. 2013, 1422–1427
9. Brakmo L S, O′Malley S W, Peterson L L. TCP vegas: new techniques for congestion detection and avoidance. In: Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM). 1994, 24–35
10. Wei D X, Jin C, Low S H, Hegde S. Fast TCP: motivation, architecture, algorithms, performance. IEEE/ACM Transactions on Networking, 2006, 14(6): 1246–1259
11. Wang J Y, Wen J T, Han Y X, Zhang J, Li C, Xiong Z. Achieving high throughput and TCP Reno fairness in delay-based TCP over large networks. Frontiers of Computer Science, 2014, 8(3): 426–439
12. Blanton E, Allman M. On making TCP more robust to packet reordering. ACM SIGCOMM Computer Communication Review, 2002, 32(1): 20–30
13. Gharai L, Perkins C, Lehman T. Packet reordering, high speed networks and transport protocol performance. In: Proceedings of the 13th International Conference on Computer Communications and Networks. 2004, 73–78
14. Zhang Z M, Guo Z Y, Yang Y Y. Bounded-reorder packet scheduling in optical cut-through switch. IEEE Transactions on Parallel and Distributed Systems, 2015, 26(11): 2927–2941
15. Zhang M, Karp B, Floyd S, Peterson L. RR-TCP: a reordering-robust TCP with DSACK. In: Proceedings of the 11th IEEE International Conference on Network Protocols. 2003, 95–106
16. Bhandarkar S, Sadry N E, Reddy A N, Vaidya N H. TCP-DCR: a novel protocol for tolerating wireless channel errors. IEEE Transactions on Mobile Computing, 2005, 4(5): 517–529
17. Wang J Y, Wen J T, Zhang J, Han Y X. TCP-FIT: a novel TCP congestion control algorithm for wireless networks. In: Proceedings of IEEE Global Communications Conference Workshops. 2010, 2065–2069
18. Piratla N M, Jayasumana A P, Bare A A. Reorder density (RD): a formal, comprehensive metric for packet reordering. In: Proceedings of International Conference on Research in Networking. 2005, 78–89
19. Zhang J, Wen J T. TCP-ACC: an active congestion compensation TCP for wireless networks. In: Proceedings of the IEEE Symposium on Computers and Communication. 2014, 1–7
20. Allman M, Paxson V, Stevens W. TCP congestion control. RFC 2581, 1999
21. Wang J Y, Wen J T, Zhang J, Han Y X. TCP-FIT: an improved TCP congestion control algorithm and its performance. In: Proceedings of the IEEE INFOCOM. 2011, 2894–2902
22. Mathis M, Mahdavi J, Floyd S, Romanow A. TCP selective acknowledgement options. RFC 2018, 1996
23. Bhatti S, Bateman M, Miras D. Revisiting inter-flow fairness. In: Proceedings of the 5th International Conference on Broadband Communications, Networks and Systems. 2008, 585–592

Jun Zhang received the BS degree in computer science and technology from Tsinghua University, China in 2010. He is currently working toward the PhD degree in computer science and technology in Tsinghua University. His research interests are in the areas of transmission control protocol, data center, and wireless network communications.

Jiangtao Wen received the BS, MS, and PhD degrees (with honors), all in electrical engineering, from Tsinghua University, China in 1992, 1994, and 1996, respectively. From 1996 to 1998, he was a staff research fellow at the University of California, Los Angeles (UCLA), USA, where he conducted cutting-edge research on multimedia coding and communications. Many of his inventions there were later adopted by international standards such as H.263, MPEG, and H.264. Since 2009, he has been a professor at the Department of Computer Science and Technology, Tsinghua University, China. He is a fellow of IEEE.

Yuxing Han received the BE degree in electrical engineering at Hong Kong University of Science and Technology (HKUST), China in 2006, and obtained her PhD degree at University of California, Los Angeles, USA in 2011. Her research interests include next generation cellular systems, cognitive radio systems, network modeling, and compressive sensing algorithms.