TCP internal buffers optimization for fast long-distance links

A. Baiocchi*, S. Mascolo†, F. Vacirca*
*Infocom Department,
University of Roma "La Sapienza",
Via Eudossiana 18, 00184 Rome, Italy.
†DEE Politecnico di Bari,
via Orabona 4, 70125 Bari, Italy.

***Abstract*— In recent years, issues regarding the behavior of TCP in high-speed and long-distance networks have been extensively addressed in the networking research community, both because TCP is the most widespread transport protocol in the current Internet and because bandwidth-delay product continues to grow. The well known problem of TCP in high bandwidth-delay product networks is that the TCP Additive Increase Multiplicative Decrease *AIMD* probing mechanism is too slow in adapting the sending rate to the end-to-end available bandwidth. To overcome this problem, many modifications have been proposed such as FAST TCP [1], STCP [2], HSTCP [3], HTCP [4], BIC TCP [5] and CUBIC TCP [6].**
The goal of this work is to investigate, by using both analytical models and simulation results, optimal sizing of the retransmission and out-of-order TCP buffers in case of modified TCP congestion control settings and to highlight differences between NewReno TCP and SACK TCP packet loss recovery schemes in terms of TCP internal buffers requirements. An important result is that the SACK option turns out to be particularly effective in reducing buffer requirements in the case of very high bandwidth-delay product links.

I. Introduction

In recent years, issues regarding the behavior of TCP in high-speed and long-distance networks have been extensively addressed in the networking research community, both because TCP is the most widespread transport protocol in the current Internet and because bandwidth-delay product continues to grow. The well known problem of TCP in high bandwidth-delay product networks is that the TCP Additive Increase Multiplicative Decrease *AIMD* probing mechanism is too slow in adapting the sending rate to the end-to-end available bandwidth.
To overcome this problem, many modifications have been proposed such as FAST TCP [1], STCP [2], HSTCP [3], HTCP [4], BIC TCP [5] and CUBIC TCP [6].
However, two key parameters are often omitted in the analysis of new congestion control proposals: they are the TCP retransmission queue size at the sender and the receiver queue that handles out-of-order packets. The first buffer is used by TCP sender to save all the outstanding packets, whereas the second one is used by the receiver entity to backlog all out-of-order received packets that cannot be withdrawn by the receiver application. These two buffers assume a fundamental role during the packet recovery phase that follows a "Fast Retransmit" retransmission, because a wrong choice of these buffer sizes can lead to remarkable underutilization of the link capacity.
The goal of this work is to investigate, by using both analytical models and simulation results, optimal sizing of the retransmission and out-of-order buffers in case of modified TCP congestion control settings and to highlight differences between NewReno TCP and SACK TCP packet loss recovery schemes in terms of TCP internal buffers requirements.
The rest of the paper is organized as follows. In Section II the TCP congestion control and the NewReno and SACK packet recovery mechanisms are described in detail. In Section III, the analysis to derive the optimal TCP internal buffer sizes is carried out, whereas in Section IV, simulative results obtained with ns-2 simulator are reported to validate analytical results. Finally, in Section V, the main conclusions are drawn.

II. Background on TCP

The basic TCP congestion control is essentially made of a probing phase and a decreasing phase. The probing phase of standard TCP consists of an exponential phase (i.e. the "Slow Start" phase) and a linear increasing phase (i.e. the "Congestion Avoidance" phase). The probing phase stops when congestion is experienced in the form of timeout or reception of $DupThresh$ duplicate acknowledgments (DUPACKs)[1]. The TCP dynamic behaviour in "steady state" condition can be considered with good approximation a sequence of congestion avoidance phases followed by reception of $DupThresh$ DUPACKs. When $DupThresh$ DUPACKs are received, the TCP implements a multiplicative decrease behavior. The generalization of the classic additive increase multiplicative decrease TCP settings can be made as follows:

a) On ACK reception:

$$cwnd \longleftarrow cwnd + a(cwnd) \quad (1)$$

b) When $DupThresh$ DUPACKs are received:

$$cwnd \longleftarrow cwnd - b \cdot cwnd \quad (2)$$

This work is supported by the Italian Ministry for University and Research (MIUR) under the PRIN project FAMOUS (http://www.tnt.dist.unige.it/famous/)

[1]The default value of $DupThresh$ is 3.

where $a(cwnd)$ is $1/cwnd$ and b is 0.5 in the case of classic TCP. The most of TCP congestion control modifications proposed for high bandwidth-delay networks can be described by modifying a and b. Some protocols (e.g. STCP) employ constant values for a and b, whereas other protocols, such as HSTCP and CUBIC, modify them dynamically. All these protocols can independently use NewReno or SACK recovery procedure independently of the congestion control algorithm. NewReno TCP and SACK TCP differ in the recovery phase, i.e. when the TCP recovers from packet losses. In particular, NewReno TCP recovery phase is based only on the cumulative ACK information whereas SACK TCP receiver exploits the TCP Selective Acknowledge option to advertise the sender about out-of-order received blocks. This information is employed by the sender to recover from multiple losses more efficiently than NewReno TCP. In the following a detailed description of NewReno and SACK loss recovery procedure is reported.

A. NewReno TCP

After that $DupThresh$ Duplicate ACKs ($DupThresh + 1$ identical ACKs requesting for the same packet sequence number) are received and that the congestion window $cwnd$ is halved and $ssthresh$ is set to $cwnd$, the requested packet is retransmitted. This phase is named "Fast Retransmit". After the "Fast Retransmit" retransmission the TCP protocol enters in a new phase named "Fast Recovery". This phase lasts till the ACK for the highest transmitted packet at the beginning of this phase (stored in the $recover$ variable) is received. During this phase, for each additional duplicate ACK[2], the congestion window is incremented by one packet, to reflect the departure from the network of an additional segment. A new segment is transmitted if it is allowed by the congestion window and the receiver advertised window. When an ACK acknowledging new packets, but with sequence number lower than $recover$ (partial ACK), the first unacknowledged packet is retransmitted, the congestion window is deflated by the amount of packet acknowledged by the partial ACK. The window deflation attempts to keep the congestion window at the level of the number of outstanding packets when the "Fast Recovery" phase ends. When an ACK acknowledging a packet greater or equal to $recover$ is received, the congestion window is set to the value of $ssthresh$ and TCP exits from the "Fast Recovery" phase and enters again the "Congestion avoidance" procedure. Figure 1 depicts a "Fast Retransmit" and "Fast Recovery" example. In this case the congestion window is 14 packets and packets 2 and 9 are lost. After 4 ACKs requesting packet 2 ($DupThresh$ = 3 DUPACKs), the sender retransmits packet 2, halves the congestion window to 7 and inflates it by one every received duplicate ACK. After 8 DUPACKs, the congestion window has grown to 15 packets and the sender is allowed to transmit a new packet every received ACK. When the retransmission of packet 2 is received by the sink, it generates a new ACK requesting for packet 9 and on the reception of this ACK, the sender is allowed to retransmit packet 9. In the meanwhile, the sender can transmits a new packet every received ACK. It is important to pinpoint the role of the sender and the receiver buffer during "Fast Recovery" and "Fast Retransmit". The sender has to store all the packets that have not been acknowledged by a cumulative ACK in the retransmission buffer and the receiver has to store in the out-of-order buffer all the packet arrived not in-sequence. In the exemple of Figure 1, for instance, the sender has to store in the retransmission buffer all the packet from sequence number 2 to 20. From the instant it retransmits packet 9 and the reception of ACK for packet 9, it has to store packets from sequence number 9 to 25. The same can be argued for the out-of-order buffer in the receiver entity, where the receiver entity has to store all the packets that have been received out-of-order before moving them in-sequence to the socket buffer where the application can pick them up.

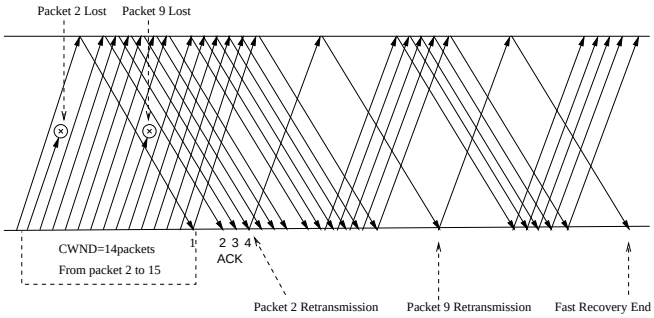


Fig. 1. NewReno Fast Retransmit and Fast Recovery example.

B. SACK TCP

With the introduction of the Selective Acknowledge information in the TCP packet header options [9], the TCP sender is informed by the receiver about non contiguous blocks of data that have been received and it can perform an accurate reconstruction of the outstanding packets and use it for lost packet retransmission. [9] specifies the format of the SACK option to include in the TCP packet header. Every block is defined by two variable that track the left and the right edge of the block of data queued at the receiver. A maximum of 4 non-contiguous blocks can be included in the TCP header options that becomes 3 when the TimeStamp option is used. Moreover [9] defines how and when the SACK information blocks are filled by the receiver entity. The SACK options should be included in all the ACKs that do not acknowledge for the sequence number of the highest received packet. The first SACK block in the TCP option must refer to block of data containing the segment that triggered the transmission of the ACK. The receiver should include the maximum number of blocks that fit into the TCP header option. These raccomandations are very important because they imply that an ACK with SACK option should be generated every time there is a hole in the sequence number space of packets in the receiver queue, and that the SACK information should be always referred to the most recent received packets.
[10] instead specifies the operations that the sender entity should perform on reception of a SACK block. After the

[2]The congestion window is inflated by $DupThresh$, taking into account that $DupThresh + 1$ ACKs are received and hence $DupThresh$ more packets that have left the network.

reception of $DupThresh$ duplicate ACKs, the sender retransmits the requested packet and starts a "Loss Recovery" phase. In this phase, the sender entity maintains the following variables: $RecoveryPoint$ is the highest sequence number of the transmitted packets; $HighACK$ is the sequence number of the highest packet cumulatively acknowledged, $HighData$ is the highest sequence number of the transmitted packets; $HighRxt$ is the highest sequence number of retransmitted packets, $Pipe$ is an estimate of the number of outstanding packets. During the "Loss Recovery" phase, on every ACK reception, the sender has to update the information of the outstanding packets (from $HighACK$ to $HighData$) in a complex data structure (scoreboard) and has to update the estimate of the $Pipe$ variable. The computation of $Pipe$ is made analyzing all the informations of the non SACKed packets in the scoreboard: the variable $Pipe$ is incremented by one for every packet below the $HighRxt$ variable and for every packet that it is not marked as lost. A packet is marked as lost when $DupThresh$ packets with a greater sequence number have been selective acknowledged.

On the reception of $DupThresh$ duplicate ACKs, $cwnd$ and $ssthresh$ are set as in the NewReno protocol, the requested packet is retransmitted, the $RecoveryPoint$ variable is set to $HighData$ and $Pipe$ is updated. The protocol enters the "Loss Recovery" phase. During this phase, for each received ACKs the sender has to update the $Pipe$ variable. If the congestion window is greater than the $Pipe$ variable, the sender is allowed to transmit a packet. The decision of the packet to transmit has to respect the following rules:

i) It has to be the packet with the smallest sequence number that it is not acknowledged selectively and that it is not already retransmitted (sequence number higher than $HighRxt$), that has been marked as lost.
ii) If no packet respects the previous rule, a new packet can be sent if the advertised window allows this.
iii) If no packet respects cases (1) and (2) it is possible to retransmit a packet with the smallest sequence number that is not acknowledged selectively and that is not already retransmitted, i.e. a packet not marked as lost that respects the same conditions of case (1).

Figure 2 depicts an example of "Fast Retransmit" and "Loss Recovery" when the SACK is implemented. As in Figure 1, two packets in the same window are lost (packets 2 and 9) and before the reception of 3 duplicate ACKs, $cwnd$ is 14 packets. TCP sender receives four cumulative ACKs that request for packet 2. After those ACKs, the TCP sender retransmits packet 2, halves the congestion window, sets the $RecoveryPoint$ to 15 and enters the "Loss Recovery" phase. Every received duplicate ACK the sender estimate the number of outstanding packets and updates the scoreboard. When the number of outstanding packets (estimated by the pipe variable) is equal to the congestion window, it can restart the transmission process. In this example, it happens that after the reception of the 8th duplicate ACK it is allowed to send a packet because the congestion window is 7 and the estimated number of outstanding packets ($Pipe$) is 6. The decision of the packet to send is performed according to the rules described previously. The sender checks if there

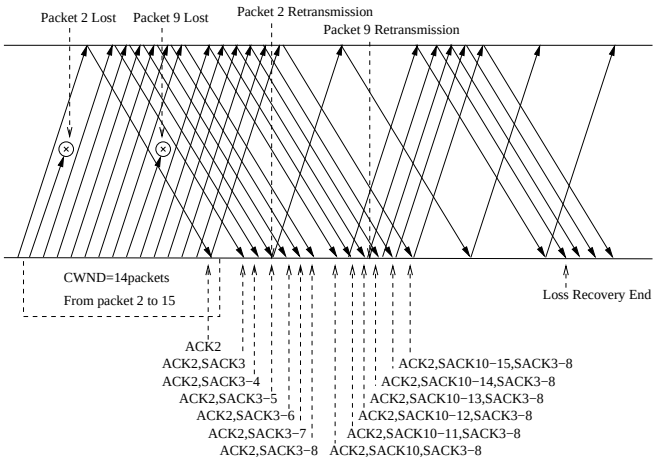


Fig. 2. TCP SACK Fast Retransmit and Loss Recovery example.

are some packets marked as lost in the scoreboard structure. No packets in the scoreboard are marked as lost (rule (1)) but the TCP sender is allowed to send a new packet (rule (2)). After the reception of the next duplicate ACK, packet 9 is marked as lost because 3 SACK blocks acknowledged for packets with sequence number greater than 9 and hence the sender is allowed to retransmit packet 9 respecting rule (1). The "Loss Recovery" phase ends when the first ACK that cumulatively acknowledges packet 15 ($RecoveryPoint$) is received.

III. Analytical model

The goal of this section is to investigate the effect of different increment and decrement factors used by congestion control algorithms and congestion control parameters in a single connection scenario. For sake of simplicity, we consider a single bottleneck scenario in the case of constant a and b values such as in the case of STCP congestion control. The considered network setting is shown in Figure 3, where B is the bottleneck buffer size, C is the link service rate (in units of packets/s), T_{fw} is the propagation delay from the TCP sender S to the bottleneck buffer, T_{fb} is the propagation delay from the bottleneck buffer to the TCP receiver R and then back to the sender. RTT_m is $T_{fb} + T_{fw}$, and RTT is the sum of RTT_m and the queuing delay in the bottleneck buffer (we are ignoring the packet transmission time $1/C$).

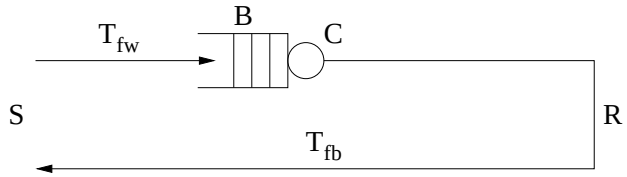


Fig. 3. Schematic of a TCP connection.

Standard TCP congestion control algorithm is made of a probing phase that increases the input rate up to fill the buffer and hit network capacity. At that point packets start to be lost and the receiver sends duplicate acknowledgments.

After the reception of $DupThresh$ DUPACKs, the sender infers network congestion and multiplicatively reduces the congestion window by b. Let t_0 be the time when the packet P being lost is transmitted, t_1 the time when the packet P is dropped at the bottleneck and t_2 the time when $DupThresh$ DUPACKs for the packet P are received by the sender. The congestion window at time t_0 is

$$W(t_0) = C \cdot RTT_m + B + 1$$

At t_1 the congestion window is increased to

$$W(t_1) = W(t_0 + T_{fw}) = W(t_0) + aCT_{fw}$$

During the interval from t_1 to t_2, $C \cdot (t_2 - t_1) = C \cdot T_{fb} + B + DupThresh$ ACKs are received and $(C \cdot T_{fb} + B) \cdot (1 + a)$ new packets are transmitted. At t_2 the congestion window is:

$$W(t_2) = W(t_1 + T_{fb} + B/C) = W(t_0) + a \cdot (CRTT_m + B)$$

After the reception of $DupThresh$ DUPACKs at time t_2^+, the packet P is retransmitted. TCP enters the recovery phase, the congestion window is reduced to:

$$W(t_2^+) = W(t_2) \cdot (1 - b)$$

and the sender stops until it gets acknowledgments for $b \cdot W(t_2)$ packets. Since the link service rate is C, the sender will stop for the interval $b \cdot W(t_2)/C$. The number of lost packets between t_0 and t_2 (excluding the packet P) is the difference between the value of the congestion window before the reception of the $DupThresh$ DUPACKS and the congestion window at t_0, i.e.:

$$W(t_2) - W(t_0) = a \cdot (CRTT_m + B)$$

If this number is greater than or equal to 1, the lost packets need to be retransmitted during the fast recovery phase; this phase finishes when all lost packets are eventually recovered. NewReno and SACK differ in the way the lost packets in the same window are recovered.

A. NewReno

During the fast recovery the sender is allowed to inflate the congestion window is order not to stop the transmissions of new packets while TCP is retransmitting the lost packets. On average, every round trip time, i.e. $RTT_m + \max(0, B - bW(t_2))/C$, (i) the transmitter sends $C \cdot RTT_m + B - bW(t_2)$ new packets and (ii) it receives the ACK for a new retransmitted packet allowing the highest acknowledged packet to be increased of $1/a + 1$ packets. Let U_n (for $n \geq 0$) be the number of unacknowledged packets when the n-th packet is retransmitted (the 0-th packet being the first lost packet P that triggered the entire recovery procedure), i.e. the difference between the highest sequence number of the transmitted packets and the highest sequence number of the acknowledged packets[3]. When P is retransmitted at t_2, U is:

$$U_0 = W(t_2)$$

[3]We are considering TCP segment sequence number instead of bytes sequence number.

and before the retransmission of the n-th lost packet (or before the reception of the ACK acknowledging the $(n-1)$-th lost packet), the number of unacknowledged packets is:

$$U_n = W(t_2) + n \cdot (C \cdot RTT_m + B - bW(t_2)) - (n-1) \cdot \left(\frac{1}{a} + 1\right) \quad (3)$$

From the previous discussion, the following observation can be derived:

i) To avoid losses during the "Fast Recovery" phase the number of outstanding packets should be less than the maximum number of packets that fits into the pipe:

$$(1-b)W(t_2) \leq B + CRTT_m \Rightarrow$$
$$(1-b) \leq \frac{1}{1 + a + 1/(CRTT_m + B)} \approx \frac{1}{1+a}$$

ii) In order to provide full link utilization, the buffer depletion time B/C must be greater or equal to the stop interval $b \cdot W(t_2)/C$:

$$\frac{B}{C} \geq \frac{bW(t_2)}{C} =$$
$$= \frac{b(CRTT_m + B + 1 + a \cdot (CRTT_m + B))}{C}$$

Assuming that the number of lost packet is small with respect to $CRTT_m + B$ (i.e. $a \ll 1$), it turns out:

$$b \leq \frac{B}{CRTT_m + B}$$

iii) The retransmission buffer and the out-of-order buffer at the receiver should be big enough to store the maximum number of unacknowledged packets. Supposing that U_n is an increasing function with n, the maximum number of unacknowledged packets is reached before the reception of the recovery ACK. This implies that the max value of U_n occurs when $n = a \cdot (CRTT_m + B)$, i.e. before the reception of the ACK acknowledging the last lost packet occurred between t_0 and t_2. Therefore it holds:

$$Q \geq U_{a(CRTT_m + B)}$$

where Q is the size of the retransmission and out-of-order buffers and the right hand side can be evaluated by letting $n = a(CRTT_m + B)$ in Eq. (3).

B. SACK TCP

After the reception of $DupThresh$ DUPACKs the SACK TCP enters in a loss recovery phase. In this phase TCP SACK is able to retransmit all the packets in a single round trip time. After the reception of $DupThresh$ DUPACKs, SACK TCP retransmits packet P and before the reception of the acknowledge for packet P, the transmitter receives in the SACK block options all the information about the $a \cdot (CRTT_m + B)$ packet losses occurred between t_0 and t_2. With this information the sender is able to retransmit all the lost packets. In this case the number of unacknowledged packets before P is retransmitted is

$$U_0 = W(t_2)$$

and before the reception of the ACK acknowledging packet P:

$$U_1 = W(t_2) + B + CRTT_m - bW(t_2) - a(CRTT_m + B)$$

because the sender will receive $B+CRTT_m$ duplicate ACKs, but it will not be able to transmit a packet for $bW(t_2)$ ACKs because the congestion window has been closed. In this round trip time the sender is able to transmit $B+CRTT_m-bW(t_2)-a\cdot(CRTT_m+B)$ new packets, since for the first $bW(t_2)$ duplicate ACKs the congestion window does not allow to send new packets, and $a\cdot(CRTT_m+B)$ duplicate ACKs with SACK information will trigger the retransmissions of packets lost after P. After the reception of the ACK acknowledging P, the number of unacknowledged packets will decrease to $(1-b)W(t_2)$ and TCP will exit from the "Loss Recovery" phase.

In the SACK TCP case, points i) and ii) for NewReno case hold, whereas point iii) should be modified as follows:

$$Q \geq U_1 = (CRTT_m + B)(2 - b - ab) + 1 - b$$

IV. Results

In the previous section, we derived the sizes of the TCP internal buffers that are necessary to store the number of unacknowledged packets U during the recovery phase. These values depend on: (a) the recovery procedure (NewReno or SACK style), (b) the link parameters and (c) the a and b settings.

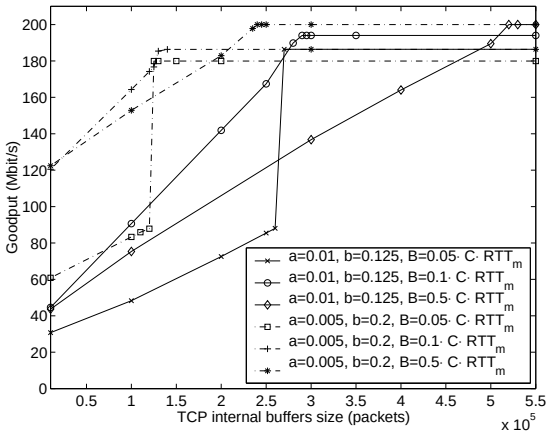
	a=0.01, b=0.125		a=0.005, b=0.2	
	NewReno	SACK	NewReno	SACK
$B = 0.05 \cdot CRTT_m$	251330	9838	120460	9445
$B = 0.1 \cdot CRTT_m$	275310	10306	131700	9895
$B = 0.5 \cdot CRTT_m$	506525	14054	239590	13493

TABLE I

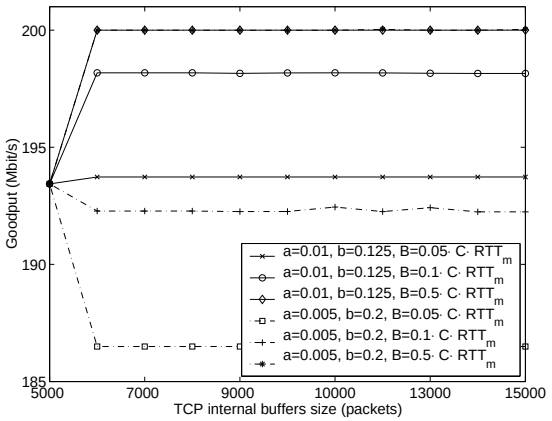
TCP INTERNAL BUFFER SIZE REQUIREMENTS.

Table I reports the values of the buffer size predicted by the model, whereas Figure 4 depicts simulation results of goodput for one NewReno TCP connection (a) and one SACK TCP connection (b) with modified values of a and b for different values of the internal buffers' size. As far as regards TCP NewReno, the model prediction is very accurate. Moreover it is possible to notice that the goodput strongly depends on the internal buffers. As for SACK TCP, it is possible to observe that the use of the SACK option greatly reduces buffer size requirements. Moreover the impact of the buffer size has a small impact on the goodput performance.

The proposed simple analytical model nicely predicts the minimum internal buffers size that is required to achieve maximum link utilization [8][4].



(a) NewReno recovery



(b) SACK recovery

Fig. 4. Goodput with C=200Mbit/s, RTT_m=300ms and packet size of 1500 bytes in the case of NewReno (a) and SACK (b) recovery procedure.

V. Conclusions

References

[1] Cheng Jin, David X. Wei and Steven H. Low, "FAST TCP: motivation, architecture, algorithms, performance," Proc. of INFOCOM 2004, March 2004, Hong Kong, China.

[2] Tom Kelly, "Scalable TCP: Improving Performance in Highspeed Wide Area Networks," Proc of PFLDnet 2003, February 2003, Geneva, Switzerland.

[3] Sally Floyd, "HighSpeed TCP for Large Congestion Windows," RFC 3649, Experimental, December 2003.

[4] R.N.Shorten, D.J.Leith, "H-TCP: TCP for high-speed and long-distance networks" Proc. PFLDnet, Argonne, 2004.

[5] L. Xu, K. Harfoush, I. Rhee, "Binary Increase Congestion Control for Fast, Long Distance Networks," Proc. of INFOCOM 2004, March 2004, Hong Kong, China.

[6] I. Rhee, L. Xu, "CUBIC: A New TCP-Friendly High-Speed TCP Variant," Proc. of PFLDnet 2005, February 2005, Lyon, France.

[7] S. Mascolo, G. Racanelli, "Testing TCP Westwood+ over Transatlantic links at 10 Gigabit/Second Rate," in proc. of PFLDnet 2005, Lyon, February 2005.

[8] S. Mascolo, F. Vacirca, "Congestion Control and Sizing Router Buffers in the Internet," in proc. of Control on Decision Conference, Sevilla, Spain, December 2005.

[9] M. Mathis, J. Mahdavi, S. Floyd, A. Romanow, "TCP Selective Acknowledgement Options," RFC 2018, October 1996.

[10] E. Blanton, M.Allman, K. Fall, L. Wang, "A Conservative Selective Acknowledgment (SACK)-based Loss Recovery Algorithm per TCP," FRC 2018, April 2003.

[4]The achievable throughput is affected by the bottleneck buffer size as well.