Available online at www.sciencedirect.com

ScienceDirect

Computer Networks 52 (2008) 1308–1327

Computer Networks

www.elsevier.com/locate/comnet

Fast retransmit and fast recovery schemes of transport protocols: A survey and taxonomy

Cheng-Yuan Ho [a,*], Yaw-Chung Chen [a], Yi-Cheng Chan [b], Cheng-Yun Ho [a]

[a] *Department of Computer Science and Information Engineering, National Chiao Tung University, No. 1001, Ta Hsueh Road, Hsinchu City 30050, Taiwan*

[b] *Department of Computer Science and Information Engineering, National Changhua University of Education, No. 1, Jin-De Road, Changhua City 50007, Taiwan*

Received 15 January 2007; received in revised form 23 October 2007; accepted 2 December 2007
Available online 1 February 2008

Responsible Editor: G. Morabito

Abstract

Although there are two standard transport protocols, TCP and UDP, offering services in the Internet, the majority of the traffic over the Internet is TCP-based. TCP-based applications can react to packet losses; however, many performance problems have been recently observed in the Internet. To resolve these problems, several new TCP fast retransmit and fast recovery algorithms have been proposed. This article surveys state-of-the-art fast retransmit and fast recovery mechanisms of TCP to address the lost packet problem, and presents a description of some useful algorithms, design issues, advantages, and disadvantages. The objective of this article is fourfold: to provide an introduction to TCP protocol; to discuss problems degrading TCP retransmission performance in the present-day Internet; to describe some proposed transport protocols that solve a number of throughput issues; and finally, to gain new insight into these protocols and thereby suggest avenues for future research. Based on our taxonomy, existing fast retransmit and fast recovery schemes of transport protocols are described in this survey.

Keywords: Fast retransmit; Fast recovery; TCP; Survey; Taxonomy

1. Introduction

The majority of the traffic over the Internet today is carried by the Transmission Control Protocol (TCP). TCP is the protocol of choice for the widely used World Wide Web (HTTP), file transfer (FTP), TELNET, and email (SMTP) applications, because it provides reliable data transport between two end hosts of a connection as well as controls the connection bandwidth usage to avoid network congestion. The behavior of TCP is therefore tightly coupled with overall Internet performance.

The essential strategy of TCP is sending packets to a network without a reservation and then react-

[*] Corresponding author. Tel.: +886 3 5731851; fax: +886 3 5714031.
E-mail addresses: cyho@csie.nctu.edu.tw (C.-Y. Ho), ycchen@csie.nctu.edu.tw (Y.-C. Chen), ycchan@cc.ncue.edu.tw (Y.-C. Chan), cyho@mail.monster.nctu.info (C.-Y. Ho).

ing to observable events. The original TCP is officially defined in [1]. It has a simple sliding window flow control mechanism without any congestion control. After observing a series of congestion collapses in 1980s, Jacobson introduced several innovative congestion control mechanisms into TCP in 1988. This TCP version, called TCP Tahoe [2], includes the slow start, additive increase and multiplicative decrease (AIMD), and fast retransmit algorithms. Two years later, the fast recovery algorithm was added to Tahoe to form a new TCP version called TCP Reno [3]. TCP Reno is currently the dominant TCP version deployed in the Internet.

Improving TCP performance is an active research area. Over the years, considerable research regarding the knowledge on TCP has been carried out [4–6]. TCP Reno can be thought of as a reactive congestion control scheme. It uses packet loss as an indicator for congestion. In order to probe the available bandwidth along the end-to-end path, the TCP congestion window is increased until a packet loss is detected, at which point the congestion window is halved and a linear increase algorithm takes over until further packet loss is experienced.

It is known that TCP Reno may periodically generate packet loss by itself and cannot efficiently recover multiple packet losses from a window of data. Moreover, the AIMD strategy of TCP Reno leads to periodic oscillations in the aspects of the congestion window size (CWND), round-trip delay, and queue length of the bottleneck node. Recent works have shown that the oscillation may induce chaotic behavior in the network, thereby adversely affecting overall network performance [7,8].

To alleviate the performance degradation problem of packet loss, many researchers attempted to refine the fast retransmit and fast recovery algorithms. New proposals included TCP NewReno [9], Forward Acknowledgment (FACK) [10], Selective Acknowledgment (SACK) [11], dynamic recovery [12], an Extension to the SACK Option (D–SACK) [13], TCP with Faster Recovery (FR–TCP) [14], Reordering–Robust TCP (RR–TCP) [15], Duplicate Acknowledgment Counting (DAC) [16], and TCP SACK+ [17]. In addition, TCP NewReno, FACK, and SACK are embedded in TCP Reno; dynamic recovery, FR–TCP, and DAC are based on TCP NewReno; and TCP DSACK, RR–TCP, and TCP SACK+ operate at the sender of SACK. All these algorithms provide performance improvement to a connection after a packet loss is detected.

Some schemes such as TCP Vegas [18,19], TCP-Peach [20], TCP-Peach+ [21], TCP-Jersey [22], and TCP Westwood [23] are called delay-based end-to-end approaches, in which the sender estimates the available network bandwidth dynamically by measuring and averaging the rate of returning acknowledgements (ACKs). TCP Vegas employs fundamentally different slow start, congestion avoidance, fast retransmit, and fast recovery approaches to combat the inherent oscillation problem of TCP Reno and to mitigate the performance degradation problem of the packet loss. TCP-Peach is particularly designed for the satellite communication environment, where a large bandwidth-delay product (BDP) is found. Amongst the four phases of TCP, TCP-Peach replaces slow start and fast recovery with sudden start and rapid recovery, respectively. In sudden start and rapid recovery, the sender probes the available network bandwidth in only one RTT with the help of low-priority dummy packets. In TCP-Peach+, the actual data packets with lower priority replace low-priority dummy packets as the probing packets to further improve the throughput. Both TCP-Jersey and TCP Westwood claim improved performance over TCP Reno, while achieving fairness and friendliness. Moreover, these two end-to-end approaches maintain the network layer structure and require minimum modification at end hosts and routers.

The taxonomy presented comprises two classification schemes: one classifies the protocols with respect to whether or not messages are exchanged between two end hosts, and the other classifies them with respect to their ability to recover multiple packet losses. This paper also provides a classification and survey of some existing protocols. Specifically, it shows how a selection of existing protocols is classified with respect to our taxonomy. Remarkably, it follows from our classification that the majority of protocols employ a relatively small set of core principles. A subset of the protocols classified is further elaborated in a survey.

The remainder of this paper is organized as follows. Section 2 discusses the problems degrading TCP retransmission performance in the present-day Internet. Our taxonomy is presented in Section 3. Section 4 provides a survey of some existing fast retransmit and fast recovery algorithms of transport protocols that solve a number of throughput issues and shows how they are classified with respect to our taxonomy. Finally, Section 5 concludes the paper with a brief summary of our proposed taxonomy and a discussion of the insight gained in developing this taxonomy.

2. Problem statements

Several problems may adversely affect connection performance when lost packets are retransmitted. We summarize these problems as follows.

2.1. Self-clocking problem

The key mechanism that follows from the Packet Conservation principle[1] is self-clocking: the sender uses returning ACKs[2] as a "clock" to determine when to send new packets into the network. Self-clocking is a crucial mechanism that keeps the data flowing on a TCP connection and protects the connection from congestion. In fact, the loss of packets is not a real problem for a TCP connection; it just indicates the occurrence of congestion in the network. However, the loss of self clocking is a serious problem, since it causes the pipeline to empty. If the pipeline empties, the TCP sender has to spend several round-trip times (RTTs) to fill the pipeline and restore the self-clocking. In other words, the loss of self-clocking severely degrades TCP performance, and must be avoided if at all possible. Furthermore, when multiple packets are lost from the same window of data, the TCP sender often incurs a retransmission timeout (RTO) and substantial performance degradation. This problem is particularly acute for Web-based document transfers, which are often so short-lived (e.g., 4–20 Kilobytes [24]) that the TCP connection never reaches steady-state before it terminates.

2.2. Slow-recovery problem

A TCP source constantly probes for available bandwidth by increasing the congestion window as long as no losses are detected. When a loss is detected, either through duplicate ACKs (dupacks) or through coarse timeout expiration, the connection backs off by shrinking its congestion window. If the loss is indicated by dupacks, TCP attempts to perform a "fast retransmit" by retransmitting the lost segments and a "fast recovery" by adjusting the congestion window. On the other hand, the congestion window is reset to its default value if the loss is followed by a coarse timeout expiration. In either case, after the congestion window is reset, the connection requires several RTTs before the window-based probing is restored to near-capacity (i.e., the slow recovery depends upon the coarse timeout expiration on long, fat pipes). This problem is exacerbated when random or sporadic losses[3] (or a combination of the two) occur. In this case a burst of lost segments is wrongfully interpreted by a TCP source as an indication of congestion, and dealt with by shrinking the sender's window. Such action, clearly, does not alleviate the random loss condition and merely results in reduced throughput. The degradation depends on the BDP.

2.3. Out-of-order-delivery problem

In today's Internet, deployment of systems that introduce packet reordering in their normal course of operation, regardless of their other benefits, may be ill-advised. This is because TCP's inability to distinguish reordering from packet loss causes the protocol to perform poorly on paths that deliver packets out of order. Losses, falsely detected or genuine, may cause TCP to send more slowly. Yet mistaking reordering for loss is not fundamental to window-based congestion control. Rather, it is an artifact of TCP's fast retransmit mechanism, which arbitrarily concludes that a packet must have been lost if it is still missing at the receiver's end after three packets sent later have arrived at the receiver. On a network path that reorders packets more than minimally, this choice of three is too aggressive in concluding loss; waiting longer before concluding loss might reveal that the packet was not lost at all, but only delayed en route.

To the extent that reordering occurs today, it is generally perceived as a transient malfunction or as an indication that a technology is maladapted for use with TCP. Route oscillation for a destination among routes with different RTTs may cause reordering [25]. Routers have been observed to

[1] This means that a new packet is not injected into the network until an old packet leaves the network. Note, however, that the Packet Conservation principle is only applied when the TCP connection is in equilibrium, with a full window of data in transit on the network (i.e., the "pipeline" is nearly fully loaded).

[2] Due to the fundamental assumption that the network does not supply any explicit feedback to the sources, today's TCP algorithms deal with network congestion through an end-to-end control algorithm.

[3] Random losses are here defined as losses not caused by congestion at the bottleneck link, as is common in the presence of wireless channels.

cease forwarding while processing a routing update, and intersperse the delayed packets with new arrivals, causing reordering [26]. The sender responds with a fast retransmit, although no actual loss has occurred. These repeated false fast retransmits keep the sender's window small and severely degrade the throughput it attains. Requiring nearly in-order delivery needlessly restricts and complicates Internet routing systems and routers. However, such beneficial systems as multi-path routing[4] and parallel packet switches[5] are difficult to deploy in a way that preserves ordering.

2.4. Retransmission-loss problem

There have been many works to avoid the RTO of TCP that takes place in an unnecessary situation. However, most current TCP implementations, even if the SACK option is used, do not have a mechanism to detect a lost retransmission and avoid subsequent RTO. In other words, the packets transmitted after the retransmission may also be lost in such a congested situation, but a lost retransmission cannot be detected without RTO if there are in sufficient ACKs (three dupacks) to trigger the fast retransmit algorithm [5]. In this time period, a TCP source may not send any new packets and the performance therefore decreases. When the subsequent retransmission timer expires, the slow start threshold (SSTHRESH) is set to one half of the current CWND and then the CWND is reset to its default value; finally, the source restarts from a slow start phase. Furthermore, a retransmission loss may lead to out-of-order delivery. A retransmission loss is not frequent in the Internet, so this problem may seem to be insignificant. Nevertheless, the loss of a retransmitted packet may actually be related with various factors such as changing level of congestion, queue management scheme, and bit error rate in a wireless channel.

[4] A TCP flow's packets are routed over multiple and/or divergent routes with distinct bottlenecks. This would increase the total end-to-end bandwidth available to the flow and cause significant packet reordering with different RTTs.

[5] A promising technique for building inexpensive high-speed routers is to use parallel forwarding and/or switching hardware. Successive packets that arrive at a router, even on the same link, may be forwarded and/or switched simultaneously by independent hardware. This simple parallel approach ignores ordering between packets processed simultaneously, and introduces reordering when packets require different processing delays.

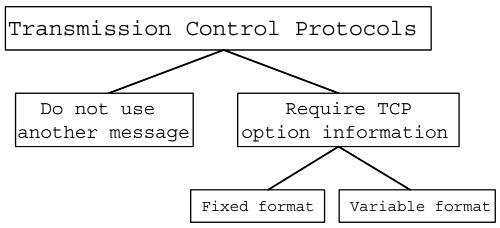
Transmission Control Protocols
Do not use another message
Require TCP option information
Fixed format
Variable format

Fig. 1. Classification with respect to attached messages.

3. The taxonomy

As mentioned in Section 1, our taxonomy consists of two classification schemes: one classifies the protocols with respect to whether or not messages are exchanged between two end hosts, and the other classifies them with respect to their ability to recover multiple packet losses. The former classification scheme is presented in Section 3.1 and the latter in Section 3.2.

3.1. Classification with respect to attached messages

Essentially, the length of a TCP header is 20 bytes if no TCP option field is used. In this article, if an end node uses the TCP option, this TCP option is called an "attached message." Fig. 1 depicts the classification scheme with respect to extra messages attached to the TCP option between two end nodes. As follows from Fig. 1, the protocols are classified into two types: *Do not use another message* and *Require TCP option information*. At the same time, the schemes that require TCP option information can be further divided in two groups depending on whether the format of the TCP option is fixed or variable.

1. *Do not use another message*: TCP source and destination do not communicate with each other by transmitting extra information attached to any TCP option. TCP Tahoe, Reno, Vegas, NewReno, Dynamic Recovery, FR–TCP, DAC, TCP-Peach, TCP-Jersey, and TCP Westwood[6] algorithms belong to this kind of transport protocol.
2. *Require TCP option information*: Extra messages, such as duplicate segments and received data blocks, are added to the TCP header by the recei-

[6] Although TCP-Jersey and TCP Westwood do not need the TCP option between a source and a destination, they use the explicit congestion notification (ECN) field of the TCP header to alert the sender of incipient congestion.

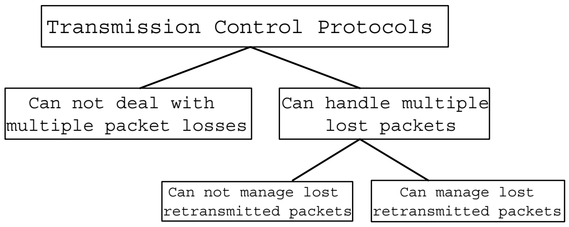
Transmission Control Protocols
Can not deal with multiple packet losses
Can handle multiple lost packets
Can not manage lost retransmitted packets
Can manage lost retransmitted packets

Fig. 2. Classification with respect to multiple packet losses.

ver to inform the sender which packets were received. For example, SACK, FACK, D–SACK, RR–TCP, SACK+, TCP-Peach+ mechanisms use the TCP option with variable formats to let the source obtain more information about successful packet delivery.

3.2. *Classification with respect to multiple packet losses*

Transport protocol schemes can be categorized into schemes that cannot deal with multiple packet losses and schemes that can handle multiple lost packets or lost retransmitted packets. A diagram showing this classification is presented in Fig. 2. A protocol that cannot deal with multiple packet drops within a single window of data operates at a very low rate and loses a significant amount of throughput. On the other hand, a TCP source may effectively utilize bandwidth and obtain high performance, because a protocol can handle multiple lost packets or lost retransmitted packets.

4. Survey of existing fast retransmit and fast recovery schemes

This section surveys a selection of the fast retransmit and fast recovery mechanisms of existing transport protocols and classifies them with respect to our taxonomy (see Section 3). Table 1 shows how the following protocols are classified: TCP Tahoe, TCP Reno, TCP Vegas, TCP NewReno, FACK, SACK, Dynamic Recovery, D–SACK, FR–TCP, RR–TCP, DAC, TCP SACK+, TCP-Peach, TCP-Peach+, TCP-Jersey, and TCP Westwood. Moreover, Fig. 3 shows the relationships between these protocols, and a classification of these mechanisms with multiple layers is plotted in Fig. 4, where type one includes TCP Tahoe and TCP Reno; type two, TCP Vegas, TCP NewReno, Dynamic Recovery, FR–TCP, TCP-Peach, TCP-Jersey, and TCP Westwood; and type three, SACK, FACK, D–SACK, RR–TCP, and TCP-Peach+. A survey of these fast retransmit and fast recovery algorithms is also presented in the following sections.

Table 1
Classification of transport protocols in our taxonomy

Protocols type	Require extra message (format)	Handle multiple packet losses	Deal with lost retransmitted packets
One	No	No	No
Two	No	Yes	No
Three	Yes (variable)	Yes	No
DAC	No	Yes	Yes
SACK+	Yes (variable)	Yes	Yes

Type one includes TCP Tahoe and TCP Reno; type two, TCP Vegas, TCP NewReno, Dynamic Recovery, FR–TCP, TCP-Peach, TCP-Jersey, and TCP Westwood; and type three, SACK, FACK, D–SACK, RR–TCP, and TCP-Peach+.

4.1. *TCP Tahoe*

Jacobson assumed that losses due to packet corruption are much less probable than losses due to buffer overflows on the network. Therefore, when a loss occurs, the sender should lower its share of the bandwidth. This is done by reducing its CWND to half the size at which the loss was found. In addition, the reasoning behind this value of one-half is that a decrease in the throughput should be equal to the multiplicative increase in the queue length in the network upon congestion.

The implementation of this multiplicative decrease is through the SSTHRESH. When a loss occurs, half the value of the CWND just before the loss is recorded in the SSTHRESH. The connection then resorts to slow start by setting the CWND to 1 packet. Slow start increases the CWND exponentially until it reaches the SSTHRESH from which it will perform an AIMD until the same thing happens again or the connection is terminated.

In order to determine that a packet is lost, Tahoe times the delay of the packet – from the sender putting a packet into the network to the time at which Tahoe receives the ACK for that packet. This value is known as the RTT. From this value (and the aggregation of timed pairs), Tahoe uses an RTO[7] to see if there is a packet loss. If an ACK is not received before this RTO, then the sender would

[7] Jacobson devised a more accurate way of estimating this value.

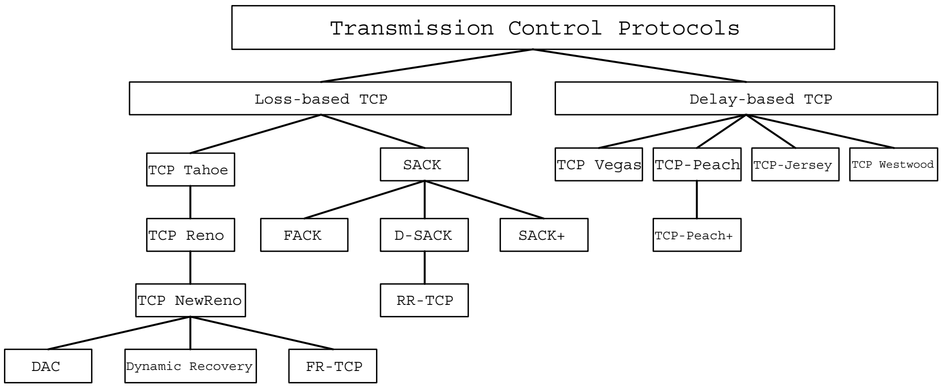

Fig. 3. Relationships between sixteen transport protocols.

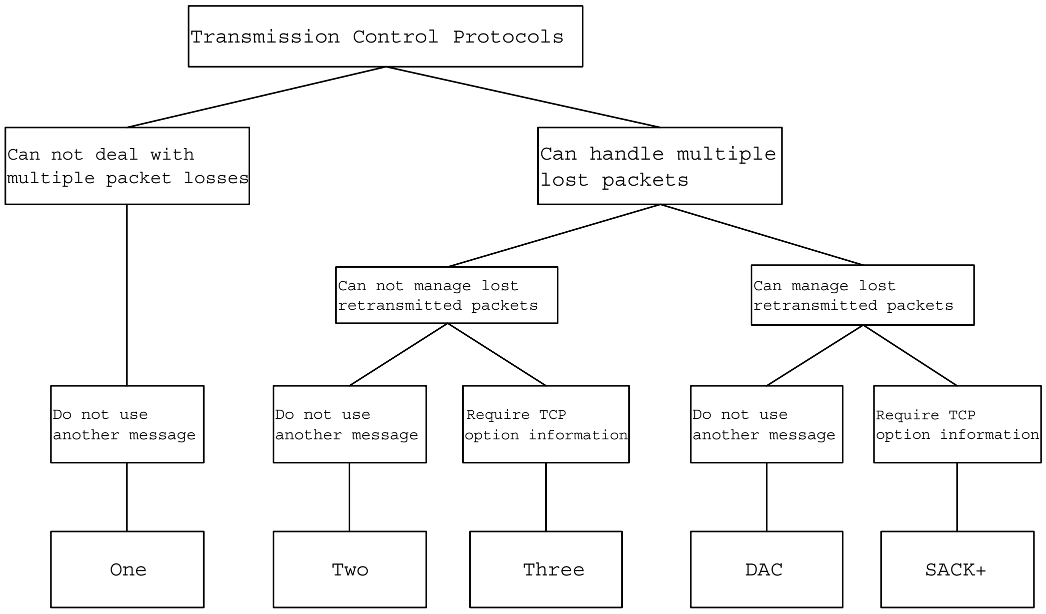
Transmission Control Protocols
Can not deal with multiple packet losses
Can handle multiple lost packets
Can not manage lost retransmitted packets
Can manage lost retransmitted packets
Do not use another message
Do not use another message
Require TCP option information
Do not use another message
Require TCP option information
One
Two
Three
DAC
SACK+

Fig. 4. Classification of sixteen protocols with multiple layers.

be confident that the packet is lost and that it should resend the packet to enable reliable delivery and movement of the window.

Another way of detecting a loss in TCP Tahoe is through the use of dupacks. Dupacks are similar to normal ACKs because they acknowledge the packets as well as tell the sender that the receiver is expecting to receive the next packet. However, the difference between a dupack and a normal ACK is that while a normal ACK acknowledges one or more previously unacknowledged packets, a dupack re-acknowledges the same packet as the previous ACK. An example of this is if a packet was lost, but all packets after the lost packets were received. Consider that a packet with the sequence number n was lost, and so the receiver could not send an ACK for it to the sender. When packet $n + 1$ arrived at the destination, the receiver told the sender that it is still expecting packet n by sending an ACK with the number n. Similarly, the receiver upon receiving packet $n + 2$, would send another ACK saying that it is still expecting packet n. These two ACKs for packet $n + 1$ and packet $n + 2$ are known as dupacks. Furthermore, a dupack does not provide the sender any new information except that the receiver is still awaiting packet n.

Typically, Tahoe waits for three dupacks before inferring this loss by the RTO, and hence immediately resends the packet. In other words, Tahoe assumes that the receipt of three dupacks actually indicates loss and then quickly retransmits the (derived) lost packet without waiting for the RTO to expire. This occurs because the RTO is relatively quite long, and it could stall the TCP transfer. This mechanism is called fast retransmit, which resends the lost packet when receiving three dupacks, sets the SSTHRESH to half the CWND, and then sets the CWND to one segment. In addition, this forces TCP to enter slow start again. TCP Tahoe can find a lost packet and retransmit it in as short a time as possible; however, it does not deal well with multiple packet drops within a single window of data.

4.2. TCP Reno

TCP Reno introduced major improvements over TCP Tahoe by changing the way in which it reacts to detecting a loss through dupacks. There are two ways in which TCP Reno detects packet loss. One is based on the reception of three dupacks, and the other is based on RTO. When a source receives three dupacks, the fast retransmit and fast recovery algorithms are performed. The source then immediately retransmits only the packet that is supposed to be lost but not subsequent ones, without waiting for a retransmission timer (also called a coarse–grained timer) to expire (the fast retransmit mechanism), and then the fast recovery mechanism is performed. (1) The slow start threshold is set to one-half the current window size. (2) The congestion window is set to the slow start threshold plus three times the packet size. (3) Each time the sender receives a dupack, it increments the congestion window by one packet and sends a new packet. (4) When the first non-duplicated ACK arrives, the congestion window is set to the slow start threshold.

If a serious congestion occurs and there are insufficient surviving packets to trigger three dupacks, the congestion will be detected by a coarse–grained retransmission timeout. When the retransmission timer expires, the slow start threshold is set to half the current congestion window size and then the congestion window size is reset to one; finally, the source restarts from the slow start phase.

The fundamental problem here is that fast retransmit algorithm assumes that only one packet was lost. This may result in loss of ACK clocking and timeouts if more than one packet are lost. Moreover, Reno encounters several problems with multiple packet losses in a window of data (usually of the order of half a window). This usually happens when invoking fast retransmit and fast recovery. Reno invokes them several times in succession, leading to multiplicative decreases in the CWND and SSTHRESH. This impacts the throughput of the connection. Further, ACK starvation may occur because of the ambiguity of dupacks and the dynamics of the CWND. This is because the sender reduces the CWND when it enters fast retransmit. The sender then receives dupacks that inflate the CWND, requiring it to send new packets until it fills its sending window in fast recovery. It then receives a non-dupack and exits fast recovery.

However, due to multiple losses in the past, this ACK will be followed by three dupacks signaling that another packet was lost; consequently, fast retransmit is entered once again after another reduction in the SSTHRESH and CWND. As this happens several times in succession, the left edge of the sending window advances only after each successive fast retransmit and the amount of data in flight (sent but not yet acked) eventually becomes more than the congestion window (halved by the latest invocation of fast retransmit). As there are no more ACKs to receive, the sender stalls and recovers from this deadlock only through a timeout, which causes a slow start.

In short, the fast retransmit and recovery algorithm of TCP Reno allows a connection to quickly recover from isolated packet losses. But when multiple packets are dropped from a window of data, TCP Reno may suffer serious performance problems. This is because it retransmits at most one dropped packet per round-trip time and further, the congestion window size may be decreased more than once due to multiple packet losses occurring during one round-trip time interval. In this situation, TCP Reno operates at a very low rate and loses a significant amount of throughput.

4.3. TCP NewReno

NewReno has modifications of the fast retransmit and fast recovery of TCP Reno. These modifications are intended to fix the Reno problems without the addition of SACK and are wholly implemented on the sender side.

In TCP Reno, partial ACKs[8] bringing the connection out of fast recovery result in a retransmission timeout in the case of multiple packet losses.

[8] Partial ACK is an acknowledgement that acknowledges some, but not all, of the outstanding packets at the start of the fast recovery phase.

In TCP NewReno, when a source receives a partial ACK, it does not exit fast recovery [5,8]. Instead, it assumes that the packet immediately following the most recently acknowledged packet has been lost and hence retransmits the lost packet. Thus, in the case of multiple packet losses, TCP NewReno will retransmit one lost packet per round-trip time until all the lost packets from the same window have been recovered, avoid requiring multiple fast retransmits from a single window of data, and not incur retransmission timeout. It remains in the fast recovery phase until all the outstanding packets at the start of that phase have been acknowledged. The implementation details of TCP NewReno are as follows:

1. *Multiple packet loss*: A fix for fast recovery to prevent starting fast retransmit and fast recovery in succession when multiple segments are dropped in the same window. So far, when entering fast retransmit, the packet with the highest sequence number was sent. NewReno performs retransmission and fast recovery algorithm, similar to Reno. However, the difference between NewReno and Reno is that when a new ACK arrives, NewReno checks if the ACK covers the highest sequence number when fast retransmit was invoked. If not, this ACK is a partial ACK and signals that another segment was lost from the same window of data. As such, NewReno retransmits the segment reported, as expected by the partial ACK and resets the retransmission timer, but does not exit fast recovery. On the other hand, if the new ACK covers the highest sequence number, NewReno will exit fast recovery but set CWND to the SSTHRESH and perform congestion avoidance.
2. *False fast recovery*: NewReno records the highest sequence number ever transmitted after a retransmission timeout. Whenever three dupacks are received, NewReno performs a test to see if it should enter fast recovery. If these ACKs cover the sequence number saved at the previous timeout, then this is a new invocation of the fast recovery. In this case, NewReno will enter fast recovery and perform the related actions. If these ACKs do not cover the sequence numbers and acknowledge the receipt of already queued segments at the receiver, NewReno will not enter fast recovery. A sender exits fast recovery only after all the outstanding packets (at the time of first loss) are acked.

NewReno makes a small change to a connection source: it may eliminate TCP Reno's wait for a retransmission timeout when multiple packets are lost from a window. Although this can avoid the unnecessary window reduction, the downside is that NewReno may take many RTTs to recover a loss episode, and it must have enough new data around to keep the ACK clock running. In other words, the recovery time of NewReno is still too long.

4.4. DAC

DAC operates at the sender side of TCP NewReno during fast recovery. The sender keeps some variables to store the expected number of dupacks for a packet loss if its retransmission is not lost again. The congestion window size just before the first fast retransmission is stored in another variable S_{cwnd}. During fast recovery, the sender counts the number of dupacks for a packet loss. If it receives more dupacks for the packet loss than the stored value, it determines that its retransmission is lost again and retransmits it without awaiting its RTO. DAC denotes the expected number of dupacks for the ith lost packet in a window by DAC_i. When the first fast retransmit is performed, the sender is not aware of how many packets are lost in a window but only knows that at least one packet is lost. Therefore, DAC_1 is always equal to $S_{\text{cwnd}} - 1$.

In order to simplify the complexity in the real Internet, there are five assumptions for the DAC algorithm:

1. Packet is assumed to be lost independently with probability p.
2. The sender has infinite packets to send so that the congestion window is always fully incremented.
3. All packets are assumed to have a same size.
4. DAC does not consider the effect of delayed acknowledgment [5] so that the receiver delivers an ACK every time a good packet is received.
5. As the size of an ACK packet is considerably small compared to a data packet, an ACK packet is assumed not to be lost.

DAC requires simple changes to TCP implementation only on the sender side and is consistent with current TCP specifications. The DAC algorithm can improve loss recovery of TCP NewReno by avoiding the unnecessary RTOs caused by retransmission losses; however, its quantitative improvement is not

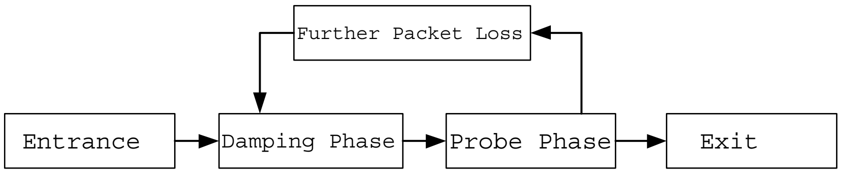

Fig. 5. Structure of Dynamic Recovery.

significant and it is a rather great improvement considering the effect of RTO on the performance of TCP.

4.5. Dynamic recovery

The dynamic recovery algorithm is intended to replace the fast recovery algorithm, which recovers lost packets when a TCP connection is congested. In addition to packet recovery, the other goal of the dynamic recovery algorithm is to probe the new equilibrium of a TCP connection during the recovery period. The changes required by the two algorithms are only on the sender side of TCP. No modifications are required on the receiver side. The structure of dynamic recovery is shown in Fig. 5.

Dynamic recovery is a congestion recovery algorithm, and it dynamically sets the threshold value that determines the end of the congestion recovery period. In addition, the threshold value varies with the number of packet losses. This threshold is set to the value of the TCP sender state variable snd.nxt at the time that the most recent packet loss was detected. The recovery phase ends as soon as snd.una advances to or beyond this threshold, which may be significantly different from the value of snd.nxt at the time the first packet loss was detected. Furthermore, in the dynamic recovery algorithm, the sender continues to send out new data packets per RTT even during the congestion recovery period. Thus, a sender not only recovers from packet losses, but also probes the new equilibrium of the connection during the congestion recovery period. The following paragraphs provide further details on the dynamic recovery algorithm.

1. *Preamble – the active number*: The key to the dynamic recovery algorithm is knowing the number of "active" data packets flowing through a connection, and not just the number of outstanding packets on a connection. In the congestion recovery phase, the outstanding packets of the connection can be divided into three groups: active, inactive and dropped. The active group is the set of data packets that are in transit, which may include retransmitted packets and packets that might be lost at the network bottleneck. The inactive group is the set of data packets that were transmitted during the past RTTs and have already arrived at the receiver, but have not yet been acknowledged[9]. The dropped group is the set of data packets that were lost during the past RTTs.Needless to say, the outstanding packets as a whole do not represent the data packets in the pipeline any more. Only the active group represents the data packets in the pipeline. Therefore, the CWND is no longer adequate for congestion control in the congestion recovery period. A new state variable actnum is thus introduced to indicate the number of active data packets in the pipeline during an RTT. At the end of the first RTT of the congestion recovery period, actnum replaces the CWND as the means to provide flow control on the sender side. Once the congestion recovery period ends, flow control responsibilities are returned to the CWND.
2. *The dynamic recovery algorithm*: As with fast recovery, dynamic recovery is triggered by a fast retransmit. However, in the dynamic recovery algorithm, the CWND is not immediately halved. Instead, it remains unchanged until the end of the recovery phase, since it is not used for congestion control in dynamic recovery.

At the beginning of the first RTT of the congestion recovery phase, the number (dupwnd) of dupacks received on the sender side (i.e., the number of data packets in the inactive group) is initialized to zero. As dupacks arrive, the sender injects two new data packets for every three received dupacks. At the same time, the state variable dupwnd counts the number of dupacks received. The end of the first RTT is indicated by the first arrival of non-dupacks.

[9] In fact, each of these packets causes the receiver to send a dupack to the sender, since these packets have sequence numbers larger than the sequence number(s) of the lost packet(s).

At the end of the first RTT of the congestion recovery period, the active number actnum assumes the role of sender flow control for the rest of the congestion recovery period. The variable actnum is initially set to dupwnd*2/3, which is the number of new data packets sent out during the first RTT of the congestion recovery period. This period is called the damping phase.

In the second phase of the dynamic recovery algorithm, called the probe phase, the sender increases the value of actnum by one[10] for every RTT, similar to the congestion avoidance algorithm. This process continues until the end of the congestion recovery phase, or until the detection of further packet losses. Since the TCP state variable snd.nxt is updated during each partial ACK, any further packet losses during the congestion recovery phase can be detected by a partial ACK that is beyond the snd.nxt value at the time the first packet loss was detected. If further packet losses occur, the dynamic recovery algorithm will reset actnum to zero and let the sender re-enter the damping subphase directly, without waiting for three further dupacks. After the sender recovers the most recently lost packet[11], it exits the congestion recovery period and enters the congestion avoidance phase. At that time, the role of flow control is transferred back to the CWND; the sender assigns the current value of actnum to the CWND and sets actnum to zero again.

From the above description, the first RTT of the congestion recovery phase is important because the role of flow control on the sender side is transferred from the CWND to the active number actnum. Note that the first lost packet is recovered in the damping subphase, and the remaining lost packets, if any, are recovered in the probe subphase. Furthermore, during the probe subphase, the sender continues to probe the new equilibrium of the TCP connection. The sending TCP can distinguish between the two phases by testing if actnum is zero. However, there is a lot of scope to improve and prove the dynamic recovery algorithm. For instance, simulation study only covered a small number of network scenarios and there is no analysis of the dynamic recovery scheme. More scenarios with different topology and traffic settings should be tested and more analytical models of performance improvement and some measurements over real networks may be definitely needed to validate the algorithm.

4.6. FR–TCP

Before discussing FR–TCP, an available bandwidth estimation (BWE) is introduced. This is because FR–TCP sets the CWND and the SSTHRESH according to the BWE value. In addition, the authors also proposed another modification of FR–TCP, called Gradual Faster Recovery TCP (GFR–TCP), to increase the output rate during fast recovery if possible.

1. *Available bandwidth estimation at the TCP source*: To overcome the lack of information on the actual available bandwidth while the congestion window is still growing, FR–TCP estimates the available bandwidth by looking at the reception rate of ACKs. In [14], the authors assumed that the TCP connection has a heavy backlog and that it suddenly experiences congestion at the bottleneck. In such conditions, it is likely that a timeout expires or three dupacks are received. In the meantime, the source has been transmitting at a rate greater than the available bandwidth. In that case, the rate of ACKs is proportional to the rate of data delivered to the destination, providing a good estimate of the (reduced) available bandwidth.If a sporadic or random loss has occurred, the rate of received ACKs is only marginally affected, and the bandwidth estimation will show little change. The basic idea is to use such an estimate of the available bandwidth to set the SSTHRESH and to compute the CWND. The rate of ACK is estimated through exponential averaging. The averaging process is run upon the reception of an ACK, including dupacks (since they signal the reception of data, although out of sequence). It is detailed by the following pseudo-code:

```
if (ACK is received) {
    sample_BWE = pkt_size*8/
    (now – lastacktime);
    BWE = BWE*alpha + sample_BWE*
    (1–alpha);
}
```

[10] Note that the dynamic recovery algorithm obeys the rule of multiplicative decrease and linear increase for the number of new data packets transmitted in the damping and probe subphases, respectively.

[11] This state is indicated by the arrival of a new ACK that is beyond the last updated snd.nxt at the time the last partial ACK arrived.

where pkt_size indicates the segment size in bytes; now, the current time, and lastacktime, the time the previous ACK was received. The parameter alpha determines the smoothing operated by the exponential filtering. In all the simulation experiments in [14], alpha = 0.8 was used. It should be noted that since the segment size is usually not fixed, the value pkt_size could be set as the average size of the last n received segments. A similar problem arises with regard to dupacks, since they do not carry information on the size of the received segment. In this case, the average size computed before the reception of the dupack is proposed for use, and the average size is updated only when new data are acked.

2. FR–TCP: This behaves like Reno as far as the sequence of actions following a triple dupacks or a coarse–grained timeout expiration are concerned. The modifications of FR–TCP to set the CWND and the SSTHRESH based on the BWE value are as follows:
   - triple dupacks:
     $SSTHRESH = (BWE^{*}RTT_{min})/a;$
     $CWND = SSTHRESH;$
   - coarse–grained timeout expiration:
     $SSTHRESH = (BWE^{*}RTT_{min})/a;$
     $CWND = 1;$

   where RTT_{min} is the smallest RTT recorded by TCP for that specific connection and a is a reduction factor. Assuming the minimum RTT excludes queuing delays, this scheme (ideally) converges to a situation where the transmission rate is equal to the actual available bandwidth between source and destination.The rationale for this strategy is quite simple. TCP Reno, after a repeated ACK or a timeout, sets the SSTHRESH to CWND/2. This translates into an output rate equal to $CWND/2^{*}RTT$ (i.e., half the output rate when the timer expired or the third dupack was received). Instead of performing this "blind" reduction of the congestion window, FR–TCP uses the estimate of the available bandwidth to set the SSTHRESH equal to a fraction $1/a$ of $BWE * RTT_{min}$. In simulations, the authors used $a = 2$, since they had experimentally verified that in the presence of one or few TCP connections, a good choice for a is 1, whereas in the presence of many TCP connections, a better choice is $a = 2$ or greater.
3. *GFR–TCP*: In slow start, TCP grabs the bandwidth rather quickly (exponentially). In contrast, in congestion avoidance, it takes a relatively long time for TCP to reach the maximum available bandwidth. If TCP experiences consecutive segment losses, the SSTHRESH becomes very small, and this leads to congestion avoidance with a very small congestion window. Subsequently, even though the network capacity may increase, TCP does not detect the bandwidth change and still widens the congestion window linearly. Thus, while there is a need for a bandwidth-aware window-decreasing algorithm (as in FR–TCP), a way to recognize when the output rate can be safely increased is also required. GFR–TCP handles the latter case.

The key idea of the GFR–TCP algorithm is twofold. Firstly, it minimizes the modifications to TCP modules, still achieving the performance equivalent to FR–TCP. Secondly, the algorithm is quite independent of the regular TCP congestion control scheme and can be applied to any TCP flavor – Tahoe, Reno, etc. The pseudo-code for the algorithm is shown below.

```
If $(CWND > SSTHRESH)$ AND $(CWND < BWE * RTT_{min})$
$SSTHRESH += (BWE * RTT_{min} - SSTHRESH)/2;$
```

The parameters above have been defined earlier. The core of the algorithm is to monitor the bandwidth in congestion avoidance and periodically increase the SSTHRESH if the conditions allow it.

According to [14], FR–TCP and GFR–TCP have two advantages: avoiding unnecessarily small windows and source-side implementation; however, the authors did not consider the friendliness and fairness toward other connections not employing these two methods in their simulations. Furthermore, refinements of the bandwidth estimation process as well as of some tuning parameters of the algorithms have not been study. Therefore, without more simulations, analysis, modeling, and real network experiments, it is hard to believe that FR–TCP and GFR–TCP solve the performance reduction problem caused by slow recovery upon a coarse timeout expiration on long, fat pipes, and the reaction to random segment losses.

4.7. SACK

Another way to deal with multiple packet losses is to tell the source which packets have arrived at the destination. SACK does so exactly. TCP adapts accumulated acknowledgement strategy to acknowledge successfully transmitted packets; this improves the robustness of acknowledgement when the path back to the source features a high loss rate. However, the drawback of accumulated acknowledgement is that after a packet loss, the source is unable to determine which packets have been successfully transmitted. Therefore, it is unable to recover more than one lost packet in each round-trip time.

The SACK option field contains a number of SACK blocks, where each SACK block reports a non-contiguous set of data that has been received and buffered. The destination uses ACK with the SACK option to inform the source that one contiguous block of data has been received out of order at the destination.

When SACK blocks are received by the source, they are used to maintain an image of the receiver queue, i.e., which packets are missing and which have been received at the destination. A scoreboard is set up to track these transmitted and received packets according to previous information in the SACK option. For each transmitted packet, the scoreboard records its sequence number and a flag bit that indicates whether the packet has been "SACKed." A packet with the SACKed bit turned on does not require to be retransmitted, but packets with the SACKed bit off and a sequence number less than the highest SACKed packet are eligible for retransmission. Whether a SACKed packet is on or off, it is removed from the retransmission buffer only when it has been cumulatively acknowledged.

SACK TCP implementation still uses the same congestion control algorithms as TCP Reno. The main difference between SACK TCP and TCP Reno is the behavior in the event of multiple packet losses. SACK TCP refines the fast retransmit and fast recovery strategy of TCP Reno so that multiple lost packets in a single window can be recovered within one round-trip time.

However, there are two shortcomings of SACK. One is that a maximum of 3 or 4 SACK blocks will be allowed in a SACK option. A SACK option that specifies n blocks will have a length of $8 \times n + 2$ bytes, so the 40 bytes available for TCP options can specify a maximum of 4 blocks. In addition, it is expected that SACK will often be used in conjunction with the Timestamp option used for round-trip time measurement (RTTM), which takes an additional 10 bytes (plus two bytes of padding); thus, a maximum of 3 SACK blocks is allowed in this case. Accordingly, if there are several non-contiguous sets of data at the destination, the sender may not know the whole situation from the SACK option. The other drawback is that if the receiver runs out of buffer space, it will discard the "SACKed" packets but not report this information to the sender. The sender will not retransmit these "SACKed" packets before it is acknowledged by the Acknowledgment Number field in the TCP header.

4.8. FACK

FACK was developed to decouple the congestion control algorithms from the data recovery algorithms. It uses the additional information provided by the SACK option to maintain an explicit measure of the total amount of outstanding data in the network. In contrast, Reno and Reno with SACK both attempt to estimate this by assuming that each dupack received represents one segment that has left the network. The FACK algorithm is able to do this in a straightforward way by introducing two new state variables, snd.fack and retran data. In addition, the sender must retain information on data blocks held by the receiver; these data blocks are required in order to use SACK information to correctly retransmit data. In addition to what is needed to control data retransmission, information on retransmitted segments must be maintained in order to accurately determine when they have left the network. The goal of FACK is to perform precise congestion control during recovery. By accurately controlling the outstanding data in the network, FACK can improve the connection throughput during the data recovery phase. The details of FACK are as follows:

At the core of the FACK congestion control algorithm is a new TCP state variable in the data sender. This new variable, snd.fack, is updated to reflect the forward-most data held by the receiver. In non-recovery states, the snd.fack variable is updated from the acknowledgment number in the TCP header and is the same as snd.una[12]. During recovery (while the receiver holds non-contiguous data), the sender continues to update snd.una from the acknowledgment number in the TCP header,

[12] The sequence number of the first byte of unacknowledged data.

but utilizes information contained in TCP SACK options to update snd.fack. When a SACK block is received that acknowledges data with a higher sequence number than the current value of snd.fack, snd.fack is updated to reflect the highest sequence number known to have been received plus one.

FACK sender algorithms that address reliable transport continue to use the existing state variable snd.una, and congestion management is altered to use snd.fack, which provides a more accurate view of the state of the network. The FACK mechanism also defines awnd to be the sender's estimate of the actual quantity of data outstanding in the network. Assuming that all unacknowledged segments have left the network:

$$\text{awnd} = \text{snd.nxt}^{13} - \text{snd.fack}.$$

During recovery, data retransmitted must also be included in the computation of awnd. The sender computes a new variable, retran_data, reflecting the quantity of outstanding retransmitted data in the network. Each time a segment is retransmitted, retran_data is increased by the segment size; when a retransmitted segment is determined to have left the network, retran_data is decreased by the segment size. Therefore, TCP's estimate of the amount of data outstanding in the network during recovery is given by

$$\text{snd.nxt} - \text{snd.fack} + \text{retran\_data}.$$

Using this measure of outstanding data, the FACK congestion control algorithm can regulate the amount of data outstanding in the network to be within one maximum segment size of the current value of the CWND:

```
while (awnd < CWND) sendsomething();
```

The FACK congestion control algorithm does not need special requirements for sendsomething(); the SACK algorithm is sufficient. Generally, sendsomething() should choose to send the oldest data first. FACK derives its robustness from the simplicity of updating its state variables: if sendsomething() retransmits old data, it will increase retran data; otherwise, it advances snd.nxt when sending new data. Correspondingly, ACKs that report new data at the receiver either decrease retran_data or increase snd.fack. Furthermore, if the sender receives an ACK that increases snd.fack beyond the value of snd.nxt at the time a segment was retransmitted (and that retransmitted segment is otherwise unaccounted for), the sender knows that the segment that was retransmitted has been lost.

Reno invokes fast recovery by counting dupacks. This algorithm causes an unnecessary delay if several segments are lost prior to receiving three dupacks. In the FACK version, the CWND adjustment and retransmission are also triggered when the receiver reports that the reassembly queue is longer than three segments. If exactly one segment is lost, the two algorithms trigger recovery on exactly the same duplicate acknowledgment. The recovery period ends when snd.una advances to or beyond snd.nxt at the time the first loss was detected. During the recovery period, the CWND is held constant; when recovery ends, TCP returns to congestion avoidance and linearly increases the CWND. In the FACK implementation, a timeout is forced if it is detected that a retransmitted segment has been lost (again). This condition is included to prevent FACK from being too aggressive in the presence of persistent congestion.

There are several unresolved issues surrounding the FACK mechanism. For example, FACK may not avoid unnecessary inflation of the congestion window through delay-sensing techniques. FACK addresses persistent congestion (when halving is not a sufficient window reduction); however, there is no clear explanation that the proposed algorithms are the best or optimal among the many possible medium window reductions. Moreover, the production Internet may lack adequate attention to issues of congestion and congestion detection. Many routers are incapable of providing full bandwidth with delay buffering and do not signal the onset of congestion through mechanisms. Although the FACK algorithm is designed to help in times of congestion, it is not a substitute for these signals in the Internet layer. The transport and internet layers must work together to improve the behavior of the Internet under high load.

4.9. D–SACK

D–SACK is an extension of the SACK mechanism. It proposes telling the sender about duplicate segments at the receiver with similar sequence numbers. As such, D–SACK requires that if a duplicate

[13] The sequence number of the first byte of unsent data.

segment is received, the receiver should send an ACK containing the duplicate data as its first SACK block. The sequence number space of the duplicate data is independent of the value of the ACK field in the TCP header. In addition, D–SACK allows the TCP sender to infer the order of packets received at the receiver when it has unnecessarily retransmitted a packet. A TCP sender could then use this information for more robust operation in an environment of reordered packets, ACK loss, packet replication, and/or early retransmit timeouts. In other words, D–SACK is a useful tool for the TCP sender to obtain more information about duplicate packets. If the receiver does not implement D–SACK, the sender will not be able to detect a false fast retransmit and behaves identically as it would in standard SACK. However, there are some drawbacks in the D–SACK mechanism. First, D–SACK cannot include all the information in its option of an ACK because of the limited TCP header space. Second, the absence of separate negotiation for D–SACK implies that the TCP receiver could send D–SACK blocks when the TCP sender does not understand this extension. In this case, the TCP sender will simply discard all D–SACK blocks and process the other blocks as it normally would. Therefore, the information in the D–SACK option may be useless to the sender.

4.10. RR–TCP

In [15], the authors propose mechanisms to detect and recover from false retransmits using the D–SACK information. This is because the D–SACK scoreboard data structure stores the per-packet state at the sender concerning recently transmitted packets and offers a natural framework for storing per-packet reordering-related information: whether a fast retransmit is false, the duration of false fast retransmit, and the reordering length a packet experiences. However, the measurement of the reordering length is more nuanced. There are two phases to sampling the distribution of reordering lengths experienced by packets: measuring the reordering length for each packet and aggregating these samples into a histogram of reordering lengths recently observed on the connection path. Therefore, the authors propose several algorithms used in RR–TCP for proactively avoiding false retransmits by adaptively varying dupthresh, which is a parameter used by fast retransmit and is fixed at three dupacks to conclude whether the network has dropped a packet. The various algorithms used are listed in Table 2.

In the DSACK–FA algorithm, the dupthresh value is chosen to avoid a percentage of false fast retransmit by setting it to the percentile value in the reordering length cumulative distribution. The percentage of reordering the algorithm avoids is known as the FA ratio.

In the DSACK–FAES algorithm, the DSACK–FA algorithm is combined with an RTT sampling algorithm that samples the RTT of retransmitted packets caused by packet delays.

The DSACK–TA algorithm uses cost functions that heuristically increase or decrease the FA ratio such that the throughput is maximized for a connection experiencing reordering. The FA ratio will increase when false retransmits occur and the FA ratio will decrease when there are significant timeouts.

In the DSACK–TAES algorithm, the DSACK–TA algorithm is combined with an RTT sampling algorithm that samples the RTT of retransmitted packets caused by packet delays.

Hence, RR–TCP can record each fast retransmit start time and the amount of window reduction in the retransmitted packet scoreboard entry. If the fast retransmit is later identified as false, RR–TCP will record the interval between the start and end of the false fast retransmit, during which period the window was unnecessarily halved. According to [15], the DSACK–TA algorithm performed the best when compared with the other algorithms. But there still remain problems for consideration:

1. To maximize throughput, a sender would like to choose paths with disjoint bottlenecks on which to send. Can the sender devise a system that allows a sender to identify paths with disjoint bottlenecks?
2. What happens when two paths have different loss rates? If the sender naively keeps a single window size, and has no knowledge of the different loss

Table 2
RR–TCP algorithms

Algorithm	Description
DSACK–FA	DSACK–R + fixed FA ratio
DSACK–FAES	DSACK–FA + enhanced RTT sampling
DSACK–TA	DSACK–FA + Timeout avoidance
DSACK–TAES	DSACK–TA + enhanced RTT sampling

characteristics of the two paths, what behavior results? Is the sender unfair (too aggressive) on either path under any circumstances? Compare the throughput the sender achieves with that achieved by two individual flows, each routed separately over one path.
3. Suppose a TCP sender is directly informed of the number of paths to be used, and the loss rate on each path. Use this information to "color" packets appropriately; marking them for which paths they should take. How close to the sum of the actual available bandwidths on each path can a sender achieve, without sending too aggressively on any one path?

4.11. TCP SACK+

Similar to DAC, TCP SACK+ detects whether a retransmitted packet is lost or not based on new packets transmitted after the retransmitted packet. Therefore, whenever a sender performs a retransmission, it stores the highest sequence number of the packets that have been already transmitted. If the retransmission of a packet loss fails, the new packets transmitted after the packet loss also deliver dupacks. In a window, SACK+ denotes the highest sequence number for the hth packet loss by S_h. If the right edge of the first SACK block included in any dupack is greater than S_h, it means that the retransmission of the hth packet loss was not successful. As a result, if there is at least a new packet transmitted after a retransmission, a sender can decide whether the retransmission is lost or not based on the information in a dupack the new packet generates.

As mentioned, SACK+ detects a lost retransmission on the basis of the well-transmitted packets after the retransmission. This means that if congestion is so heavy that no packet should be sent, SACK+ does not transmit additive packets by detecting a lost retransmission. This is because the packets transmitted after the retransmission may also be lost in such a congested situation. However, a retransmission loss detected assures that there has been a heavy congestion. Therefore, it should be considered as two indications of congestion; this leads to decreasing the congestion window twice, as specified in [27].

TCP SACK+ can be implemented with simple changes only to the sender part of TCP and it is consistent with the conservative principle of window management of TCP. However, a retransmission loss is not frequent in the real Internet; therefore, the improvement of TCP SACK+ may seem to be insignificant.

4.12. TCP Vegas

In TCP Vegas, as in TCP Reno, a triple-duplicate ACK always results in packet retransmission. However, in order to retransmit the lost packets quickly, Vegas extends Reno's fast retransmission strategy. Vegas measures the RTT for every packet sent based on fine–grained clock values. Using these fine–grained RTT measurements, a timeout period for each packet is computed. When a duplicate ACK is received, Vegas checks whether the timeout period of the oldest unacknowledgement packet has expired. If so, the packet is retransmitted. This modification leads to packet retransmission after just one or two duplicate ACKs. When a non-duplicate ACK that is the first or second ACK after a fast retransmission is received, Vegas again checks for the expiration of the timer and may retransmit another packet. Note that packet retransmission due to an expired fine–grained timer is conditioned on receiving certain ACKs. This technique enables the faster detection of losses and recovery from multiple drops without restarting the slow start phase if the retransmission timer does not expire before. Hence, it allows dealing with a problem that Reno suffers from considerable, namely, multiple drops in the same data window.

After a packet retransmission is triggered by a dupack and the ACK of the lost packet is received, the congestion window size is reduced to alleviate the network congestion. There are two cases for Vegas to set the congestion window size. If the lost packet has been transmitted just once, the congestion window size will be three fourth of the previous size. Otherwise, Vegas considers the congestion to be more serious, and one-half of the previous congestion window size is set into the current congestion window. Notably, in the case of multiple packet losses occurring during one RTT and triggering more than one fast retransmission, the congestion window is reduced only for the first retransmission.

If a loss episode is severe enough that no ACKs are received to trigger the fast retransmit algorithm, eventually, the losses are identified by a Reno-style coarse–grained timeout. When this occurs, the slow start threshold is set to one-half of the congestion window size; then, the congestion window is reset to two and finally, the connection restarts from slow start. The flowchart of TCP Vegas is shown in Fig. 6.

4.13. TCP-Peach

TCP-Peach, a congestion control scheme proposed for satellite networks, uses dummy segments (that must be treated as low-priority segments by all intermediate nodes) to probe the availability of network resources. If all the dummy segments are acknowledged, then the sender interprets this as evidence that there are unused resources in the network and accordingly can chosen to increase its transmission rate. In TCP-Peach, corruption errors are not explicitly notified, but instead implicitly accounted for by the capacity estimation strategy. TCP-Peach contains the following algorithms: sudden start, congestion avoidance, fast retransmit, and rapid recovery. The congestion avoidance and fast retransmit algorithms may be those proposed either in TCP Reno or by TCP Vegas. TCP-Peach's flowchart is shown in Fig. 7[14].

The rapid recovery first maintains the classical fast recovery conservative assumption that all segment losses are due to network congestion because the TCP layer does not know anything about the exact causes for the losses, i.e., due to network congestion or link errors. Accordingly, the sender halves its CWND, as in TCP Reno. In order to probe the availability of network resources, the sender transmits a certain number n_{Dummy} of dummy segments. Note that the value for n_{Dummy} will be derived subsequently. The ACKs for the dummy segments will be received after the ACK for the lost data segment, i.e., they will be received when the sender is in the congestion avoidance phase.

If the segment loss is due to congestion, then the congested router can serve CWND segments approximately per round-trip time. As a result, the network will accommodate CWND/2 data segments, which have high priority, and CWND/2 dummy segments out of the n_{Dummy} dummy segments transmitted during the rapid recovery phase. Therefore, the sender must not increase its congestion window when it receives the first CWND/2 ACKs for dummy segments. In fact, these ACKs cannot be considered as a sign that the loss was due to link errors. This will prevent the sender from increasing its congestion window when the first CWND/2 ACKs for dummy segments are received during congestion avoidance. After receiving CWND/2 ACKs for dummy segments, the sender increases its congestion window by one segment each time it receives an ACK for a dummy segment. If all dummy segments are ACKed to the sender, the congestion window reaches its value before the segment loss was detected. Note that the retransmitted segment may get lost. Let t_{Retr} be the time when the lost segment is retransmitted. If at time $(t_{\text{Retr}} + \text{RTO})$, no ACK has been received for the retransmitted segment, this segment may be lost. Accordingly, the rapid recovery is terminated and the sender executes the sudden start because the loss may be due to heavy network congestion.

In short, TCP-Peach is based on the replacement of slow start and fast recovery algorithms with sudden start and rapid recovery procedures, which rely on the introduction of dummy segments to probe the bandwidth availability of the network. TCP-Peach requires all the routers along the connection to implement some priority mechanism at the IP layer so as to discard dummy segments in the presence of congestion.

4.14. TCP-Peach+

In TCP-Peach, dummy segments are transmitted in rapid recovery to resume the CWND rapidly from decrement due to segment loss caused by a link error. Although it solves the problem of throughput degradation in satellite networks over Fast Recovery, when the link error is high and multiple segment losses occur within one window of data, the throughput degradation is still large. Therefore, quick recovery is proposed to recover from high link errors.

Similar to the case of TCP SACK, the authors adopted the SACK option field in TCP-Peach+ to avoid retransmitting out of order segments received at the destination. At the sender, a data structure called a scoreboard is maintained to update information about cumulatively ACKed and SACKed segments. During quick recovery, a variable called pipe that represents the estimated number of segments outstanding in the network is maintained. The variable pipe is incremented by one when the sender either sends a new segment or retransmits an old segment. It is decreased by one when the sender receives an ACK that reports new data has been received at the receiver and has left the pipe. When-

[14] The flowchart of TCP-Peach+ is the same as that of TCP-Peach. The difference between TCP-Peach and TCP-Peach+ is presented in the next section.

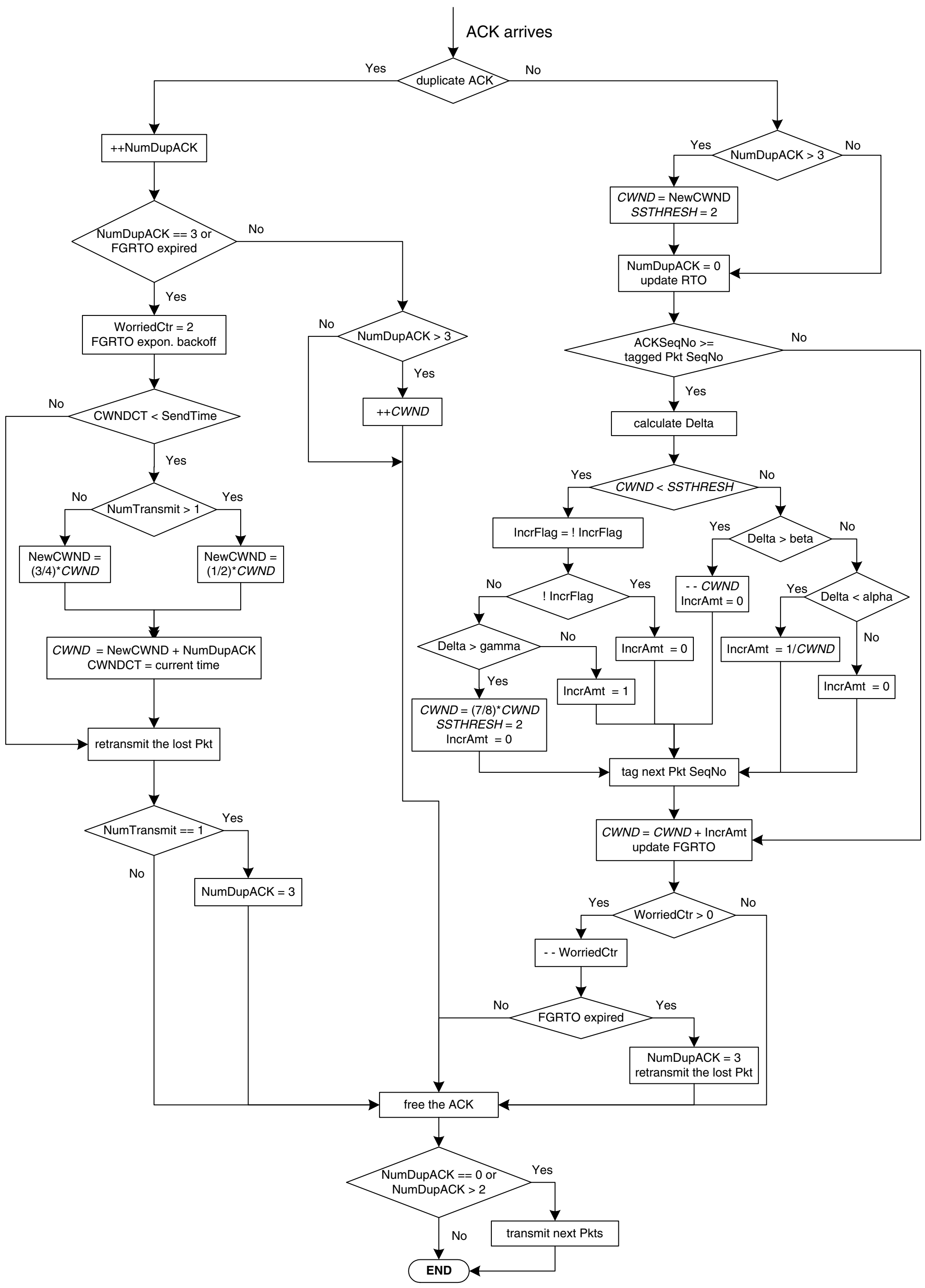

Fig. 6. Flowchart of TCP Vegas.

ever a SACK arrives, the retransmit timer is also reset. When the ACK for the segment arrives right before quick recovery is entered, ACKing all data that is outstanding before quick recovery, the sender exits quick recovery and begins congestion avoidance normally.

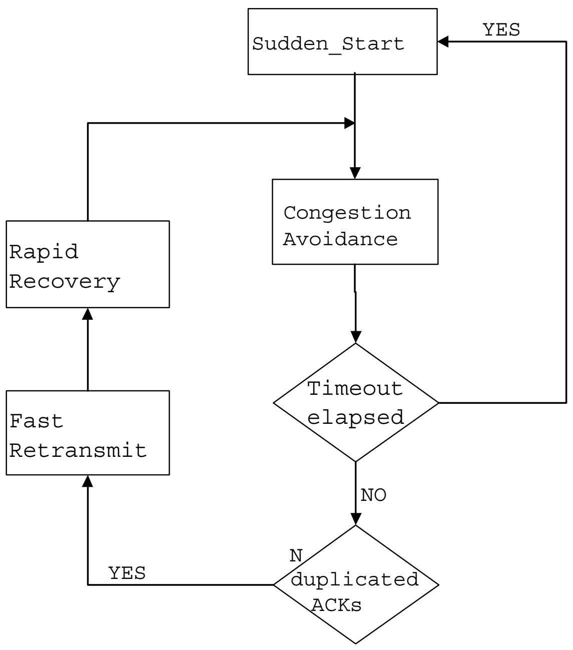

Fig. 7. Flowchart of TCP-Peach.

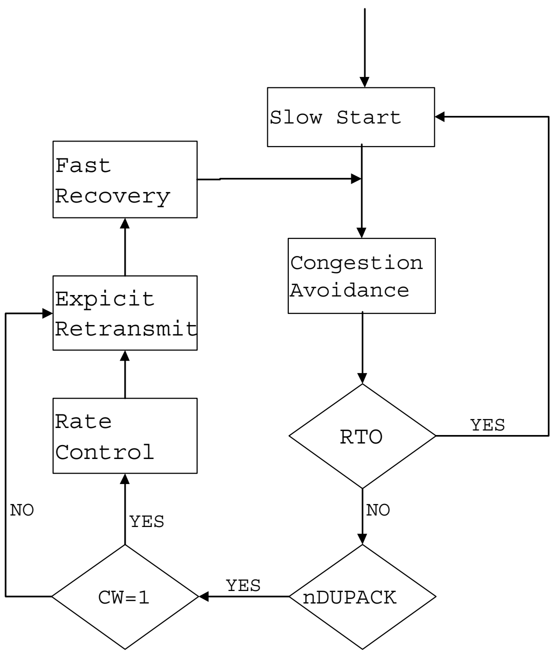

Fig. 8. The flow diagram of TCP-Jersey.

4.15. TCP-Jersey

TCP-Jersey adopts slow start, congestion avoidance, and fast recovery from Reno but replaces Reno's fast retransmit with explicit retransmit and introduces the rate control procedure. The flow diagram of TCP-Jersey is shown in Fig. 8. The only difference between Reno's fast retransmit procedure and Jersey's explicit retransmit procedure is that unlike Reno's retransmit procedure, which halves the current congestion window before starting the retransmission, explicit retransmit maintains the current CWND. It leaves the adjustment of the congestion window to the rate control procedure. The operation of the rate control procedure is also quite simple. The procedure sets the SSTHRESH to the optimum congestion window size computed based on its available bandwidth estimator (ABE), and sets the CWND to the SSTHRESH if the connection is in the congestion avoidance phase. The sender receiving a module in TCP-Jersey operates as follows. Upon entry, it invokes the ABE procedure. If an ACK is received without the congestion warning (CW) mark, it proceeds as Reno (i.e., invoking slow start or congestion avoidance depending on whether or not the CWND is below the SSTHRESH). If the received ACK or the *n*th dupack is marked with the CW bit, it calls the rate control procedure to adjust the window size and proceeds with slow start or congestion avoidance if it is an ACK; otherwise, it enters the explicit retransmit if it is the *n*th dupack. When the *n*th dupack is received without the CW mark, TCP-Jersey renders the packet drop caused by a random error and therefore enters explicit retransmit without adjusting the window size.

4.16. TCP Westwood

In TCP Westwood, the congestion window increments during slow start and congestion avoidance remain the same as in Reno, that is, they are exponential and linear, respectively. A packet loss is indicated by (a) the reception of 3 dupacks or (b) coarse timeout expiration. In case the loss indication is 3 dupacks, TCP Westwood sets the CWND and SSTHRESH as follows:

```
if (3 dupacks are received) {/*congestion avoid */
    SSTHRESH = (BWE*$RTT_{min}$)/seg_size;
    if (CWND > SSTHRESH) CWND = SSTHRESH;
}
```

In the pseudo-code, seg_size denotes the length of a TCP segment in bits. Note that the reception of *n* dupacks is followed by the retransmission of the missing segment, as in the standard Fast Retransmit implemented by TCP Reno. In addition, the window growth after the CWND is reset to the SSTHRESH according to the rules established in the Fast Retransmit algorithm. During the congestion avoidance phase, TCP Westwood is probing for extra available bandwidth. Therefore, when *n* dupacks are received, it means that TCP Westwood has hit the network capacity (or that in the case of wireless links, one or more segments were dropped due to sporadic losses). Thus, the SSTHRESH is set equal to the window capable of producing the measured rate BWE when the bottleneck buffer is empty. The congestion window is set equal to the SSTHRESH and the congestion avoidance phase is entered again to gently probe for new available bandwidth. Note that after the SSTHRESH has been set, the congestion window is set equal to the slow start threshold only if CWND > SSTHRESH. It is possible that the current CWND may be below the threshold. This occurs after time-out for example, when the window is dropped to one. During slow start, the CWND still features an exponential increase as in the current implementation of TCP Reno.

In case a packet loss is indicated by a timeout expiration, the CWND and SSTHRESH are set as follows:

```
if (coarse timeout expires){
    CWND = 1;
    SSTHRESH = $(BWE^{*}RTT_{min})$/seg_size;
    if (SSTHRESH < 2) SSTHRESH = 2;
}
```

The rationale for the algorithm above is that after a timeout, the CWND and the SSTHRESH are set equal to one and BWE, respectively. Thus, the basic Reno behavior is still captured, while a speedy recovery is ensured by setting the SSTHRESH to the BWE.

5. Conclusion

This paper presents a taxonomy for fast retransmit and fast recovery mechanisms of some existing transport protocols. The taxonomy comprises two classification schemes. The first scheme classifies protocols with respect to extra messages attached to the TCP option between two end nodes. In this scheme, protocols are classified into two types: *do not use another message* and *require TCP option information*. The second classification scheme classifies protocols with respect to their retransmission behavior when multiple packets are lost. Therefore, transport protocol schemes can be categorized into schemes that cannot deal with multiple packet losses and those that can handle multiple lost packets or lost retransmitted packets. The article also shows how existing protocols are classified according to this taxonomy and surveys a subset of the classified protocols.

A good fast retransmit algorithm should distinguish between a real lost packet from an out-of-order delivery packet and perform a retransmission of what appears to be the missing segment as soon as possible, without waiting for a retransmission timer to expire; on the other hand, a good fast recover scheme should allow high throughput under light or moderate congestion, especially for large windows; further, it should increase the output rate safely when the bandwidth has available resources. Hence, the proposal of a simple and efficient fast retransmit and fast recovery algorithms is not easy. Moreover, TCP is widely used over all kinds of networks such as wireless, Wi-Fi, and WiMAX. Fast retransmit and fast recovery algorithms may face new challenges in these networks. Therefore, there is still scope for improving fast retransmit and fast recovery algorithms.

In summary, this paper presents a taxonomy for fast retransmit and fast recovery mechanisms of some existing transport protocols. This taxonomy provides a unified terminology and a framework for the comparison and evaluation of this class of protocols. In addition, the insight provided by the taxonomy and survey in this paper may be used to guide future research in this area. Studying TCP behavior is still an active area of research and requires further investigation since several mechanisms beside the ones described in this article rely on this type of estimation. Simulation results obtained by comparing different protocols and a theoretical study of the transport protocols will be discussed and developed in the forthcoming article.

References

[1] J. Postel, Transmission Control Protocol, in: IETF RFC 793 (September) (1981).
[2] V. Jacobson, Congestion avoidance and control, in: Proceedings of ACM SIGCOMM, 1988, pp. 314–329.
[3] V. Jacobson, Modified TCP congestion avoidance algorithm, Technical Report, 1990.
[4] J. Padhye, V. Firoiu, D. Towsley, J. Kurose, Modeling TCP throughput: a simple model and its empirical validation, in: Proceedings of ACM SIGCOMM, 1998, pp. 303–314.

[5] K. Fall, S. Floyd, Simulation-based comparisons of Tahoe, Reno, and SACK TCP, ACM Computer Communication Review 26 (3) (1996) 5–21.
[6] J. Mo, R.J. La, V. Anantharam, J. Walrand, Analysis and comparison of TCP Reno and Vegas, in: Proceedings of IEEE INFOCOM, 1999, pp. 1556–1563.
[7] W. Feng, P. Tinnakornsrisuphap, The failure of TCP in high-performance computational grids, in: Proceedings of SC 2000: High-performance Networking and Computing Conference, 2000.
[8] A. Veres, M. Boda, The chaotic nature of TCP congestion control, in: Proceedings of IEEE INFOCOM, 2000, pp. 1715–1723.
[9] J.C. Hoe, Start-up dynamics of TCP's congestion control and avoidance schemes, Master's Thesis, MIT, 1995.
[10] M. Mathis, J. Mahdavi, Forward acknowledgement: refining TCP congestion control, in: Proceedings of ACM SIGCOMM, 1996, pp. 181–191.
[11] M. Mathis, J. Mahdavi, S. Floyd, A. Romanow, TCP selective acknowledgement options, IETF RFC 2018 (October) (1996).
[12] H. Wang, C.L. Williamson, A new scheme for TCP congestion control: smooth-start and dynamic recovery, in: Proceedings of IEEE MASCOTS, 1998, pp. 69–76.
[13] S. Floyd, J. Mahdavi, M. Mathis, M. Podolsky, An extension to the selective acknowledgement (SACK) option for TCP, IETF RFC 2883 (July) (2000).
[14] C. Casetti, M. Geria, S.S. Lee, S. Mascolo, M. Sanadidi, TCP with faster recovery, in: Proceedings of IEEE MILCOM, vol. 1, 2000, pp. 320–324.
[15] M. Zhang, B. Karp, S. Floyd, L.L. Peterson, RR–TCP: a reordering–robust TCP with DSACK, in: Proceedings of IEEE ICNP, 2003, pp. 95–106.
[16] B. Kim, J. Lee, Retransmission loss recovery by duplicate acknowledgment counting, IEEE Communications Letters 8 (1) (2004) 69–71.
[17] B. Kim, D. Kim, J. Lee, Lost retransmission detection for TCP SACK, IEEE Communications Letters 8 (9) (2004) 600–602.
[18] L.S. Brakmo, S.W. O'Malley, L.L. Peterson, TCP Vegas: new techniques for congestion detection and avoidance, in: Proceedings of ACM SIGCOMM, 1994, pp. 24–35.
[19] L.S. Brakmo, L.L. Peterson, TCP Vegas: end to end congestion avoidance on a global internet, IEEE Journal on Selected Areas in Communication 13 (1995) 1465–1480.
[20] I. Akyildiz, G. Morabito, S. Palazzo, TCP-Peach: a new congestion control protocol for satellite IP networks, IEEE/ACM Transactions on Networking 9 (3) (2001) 307–321.
[21] I.F. Akyildiz, X. Zhang, J. Fang, TCP Peach+: enhancement of TCP Peach for satellite IP networks, IEEE Communications Letters 6 (7) (2002) 303–305.
[22] K. Xu, Y. Tian, N. Ansari, TCP-Jersey for wireless IP communications, IEEE Journal on Selected Areas in Communications 22 (4) (2004) 747–756.
[23] TCP-Westwood, URL: http://www.cs.ucla.edu/NRL/hpi/tcpw/.
[24] M.F. Arlitt, C.L. Williamson, Internet web servers: workload characterization and performance implications, IEEE/ACM Transactions on Networking 5 (5) (1997) 631–645.
[25] V. Paxson, End-to-end routing behavior in the Internet, in: Proceedings of ACM SIGCOMM, vol. 26, 1996, pp. 25–38.
[26] V. Paxson, End-to-end Internet packet dynamics, in: Proceedings of ACM SIGCOMM, vol. 27, 1997, pp. 139–152.
[27] M. Allman, V. Paxson, W. Stevens, TCP congestion control, IETF RFC 2581 (April) (1999).

Cheng-Yuan Ho is currently a Ph.D. candidate in Computer Science at National Chiao Tung University, Hsinchu City, Taiwan. He also works with the Wireless and Networking Group of Microsoft Research Asia, Beijing, China since December, 2005. His research interests include the design, analysis, and modelling of the congestion control algorithms, high speed networking, QoS, and mobile and wireless networks.

Yaw-Chung Chen received his Ph.D., degree in computer science from Northwestern University, Evanston, Illionis, USA in 1987. During 1987–1990, he worked at AT&T Be Laboratories. Now he is a professor in the Department of Computer Science of National Chiao Tung University. His research interests include multimedia communications, high speed networking, and wireless networks.

Yi-Cheng Chan received his Ph.D., degree in Computer Science and Engineering from National Chiao Tung University, Taiwan in 2004. He is now an assistant professor in the department of computer science and information engineering of National Changhua University of Education, Changhua City, Taiwan. His research interests include network protocols, wireless networks, and AQM.

Cheng-Yun Ho is currently a M.S. student in Computer Science at National Chiao Tung University, Hsinchu City, Taiwan. He received his B.S. degree in Computer Science at National Cheng Chi University, Taipei City, Taiwan. His research interests include mobile IP, network protocols, and sensor network.