Host-to-Host Congestion Control for TCP

Alexander Afanasyev, Neil Tilley, Peter Reiher, and Leonard Kleinrock
University of California, Los Angeles
{afanasev, tilleyns, reiher, lk}@cs.ucla.edu

***Abstract*—The Transmission Control Protocol (TCP) carries most Internet traffic, so performance of the Internet depends to a great extent on how well TCP works. Performance characteristics of a particular version of TCP are defined by the congestion control algorithm it employs. This paper presents a survey of various congestion control proposals that preserve the original host-to-host idea of TCP—namely, that neither sender nor receiver relies on any explicit notification from the network. The proposed solutions focus on a variety of problems, starting with the basic problem of eliminating the phenomenon of congestion collapse, and also include the problems of effectively using the available network resources in different types of environments (wired, wireless, high-speed, long-delay, etc.). In a shared, highly distributed, and heterogeneous environment such as the Internet, effective network use depends not only on how well a single TCP-based application can utilize the network capacity, but also on how well it cooperates with other applications transmitting data through the same network. Our survey shows that over the last 20 years many host-to-host techniques have been developed that address several problems with different levels of reliability and precision. There have been enhancements allowing senders to detect fast packet losses and route changes. Other techniques have the ability to estimate the loss rate, the bottleneck buffer size, and level of congestion. The survey describes each congestion control alternative, its strengths and its weaknesses. Additionally, techniques that are in common use or available for testing are described.**

***Index Terms*—TCP, congestion control, congestion collapse, packet reordering in TCP, wireless TCP, high-speed TCP**

I. Introduction

Most current Internet applications rely on the Transmission Control Protocol (TCP) [1] to deliver data reliably across the network. Although it was not part of its initial design, the most essential element of TCP is congestion control; it defines TCP's performance characteristics. In this paper we present a survey of the congestion control proposals for TCP that preserve its fundamental host-to-host principle, meaning they do not rely on any kind of explicit signaling from the network.[1] The proposed algorithms introduce a wide variety of techniques that allow senders to detect loss events, congestion state, and route changes, as well as measure the loss rate, the RTT, the RTT variation, bottleneck buffer sizes, and congestion level with different levels of reliability and precision.

The key feature of TCP is its ability to provide a reliable, bi-directional, virtual channel between any two hosts on the Internet. Since the protocol works over the IP network [3], which provides only best-effort service for delivering packets across the network, the TCP standard [1] specifies a sliding window based flow control. This flow control has several mechanisms. First, the sender buffers all data before the transmission, assigning a sequence number to each buffered byte. Continuous blocks of the buffered data are packetized into TCP packets that include a sequence number of the first data byte in the packet. Second, a portion (window) of the prepared packets is transmitted to the receiver using the IP protocol. As soon as the sender receives delivery confirmation for at least one data packet, it transmits a new portion of packets (the window slides along the sender's buffer, Figure 1). Finally, the sender holds responsibility for a data block until the receiver explicitly confirms delivery of the block. As a result, the sender may eventually decide that a particular unacknowledged data block has been lost and start recovery procedures (e.g., retransmit one or several packets).

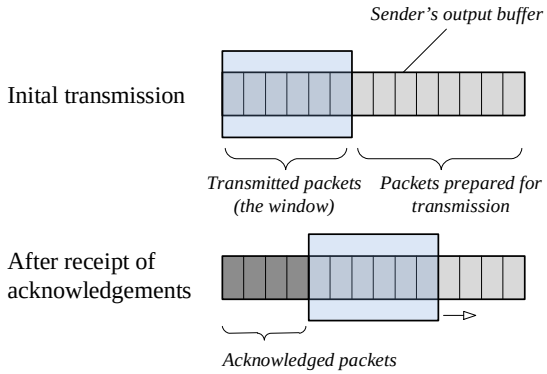


Fig. 1. Sliding window concept: the window slides along the sender's output buffer as soon as the receiver acknowledges delivery of at least one packet

To acknowledge data delivery, the receiver forms an ACK packet that carries one sequence number and (optionally) several pairs of sequence numbers. The former, a *cumulative ACK*, indicates that all data blocks having smaller sequence numbers have already been delivered. The latter, a *selective ACK* (Section II-E—a TCP extension standardized 15 years after the introduction of TCP itself), explicitly indicates the ranges of sequence numbers of delivered data packets. To be more precise, TCP does not have a separate ACK packet, but rather uses flags and option fields in the common TCP header for acknowledgment purposes. (A TCP packet can be both a data packet and an ACK packet at the same time.) However, without loss of generality, we will discuss a notion of ACK packets as a separate entity.

Although a sliding window based flow control is relatively simple, it has several conflicting objectives. For example, on the one hand, throughput of a TCP flow should be maximized.

[1]Lochert et al. [2] have presented a thorough survey on the congestion control approaches which rely on explicit network signaling.

This essentially requires that the size of a sliding window also be maximized. (It can be shown that the maximum throughput of a TCP flow depends directly on the sliding window size and inversely on the round-trip time of the network path.) On the other hand, if the sliding window is too large, there is a high probability of packet loss because the network and the receiver have resource limitations. Thus, minimization of packet losses requires minimizing the sliding window. Therefore, the problem is finding an optimal value for the sliding window (which is usually referred to as the *congestion window*) that provides good throughput, yet does not overwhelm the network and the receiver.

Additionally, TCP should be able to recover from packet losses in a timely fashion. This means that the shorter the interval between packet transmission and loss detection, the faster TCP can recover. However, this interval cannot be too short, or otherwise the sender may detect a loss prematurely and retransmit the corresponding packet unnecessarily. This overreaction simply wastes network resources and may induce high congestion in the network. In other words, when and how a sender detects packet losses is another hard problem for TCP.

The initial TCP specification [1] is designed to guard only against overflowing the input buffers at the receiver end. The incorporated mechanism is based on the *receiver's window* concept, which is essentially a way for the receiver to share the information about the available input buffer with the sender. Figure 2 illustrates this concept in schematic fashion. When establishing a connection, the receiver informs the sender about the available buffer size for incoming packets (in the example shown, the receiver's window reported initially is 8). The sender transmits a portion (window) of prepared data packets. This portion must not exceed the receiver's window and may be smaller if the sender is not willing (or ready) to send a larger portion. In the case where the receiver is unable to process data as fast as the sender generates it, the receiver reports decreasing values of the window (3 and 1 in the example). This induces the sender to shrink the sliding window. As a result, the whole transmission will eventually synchronize with the receiver's processing rate.

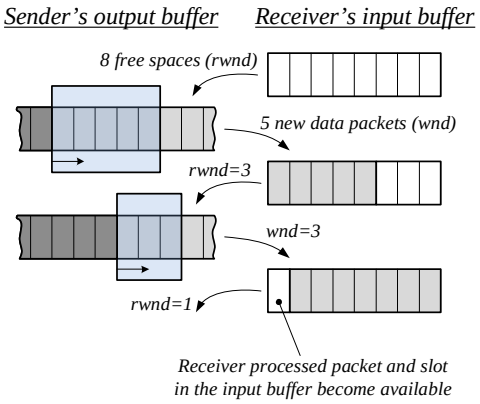


Fig. 2. Receiver's window concept: receiver reports a size of the available input buffer (receiver's window, $rwnd$) and sender sends a portion (window, wnd) of data packets that does not exceed $rwnd$

Unfortunately, protocol standards that remain unaware of the network resources have created various unexpected effects on the Internet, including the appearance of congestion collapse (see Section II). The problem of congestion control, meaning intelligent (i.e., network resource-aware) and yet effective use of resources available in packet-switched networks, is not a trivial one. In this survey we collected and classified a number of the proposed congestion control algorithms that optimize various parameters of TCP data transfer without relying on any explicit notifications from the network. In other words, they preserve the host-to-host principle of TCP, whereby the network is seen as a black box.

Section II is devoted to congestion control proposals that build a foundation for all currently known host-to-host algorithms. This foundation includes 1) the basic principle of probing the available network resources, 2) loss-based and delay-based techniques to estimate the congestion state in the network, and 3) techniques to detect packet losses quickly. However, the techniques that are developed are not universal. For example, Tahoe's initial assumption that packets are not generally reordered during transmission may be wrong in some environments. As a result, the performance of Tahoe flows in these environments will prove inadequate (Section II-A). In Section III we discuss congestion control proposals that modify previously developed algorithms to tolerate various levels of packet reordering.

As data transfer technologies and the Internet itself have evolved, the research focus for congestion control algorithms has been changing from basic congestion to more sophisticated problems. In Section IV we review the network resource optimization problem. In particular, we discuss two algorithms which discover the ability of a TCP congestion control to provide traffic prioritization in a pure host-to-host fashion.

In Section V we discuss congestion control algorithm proposals which try to improve the performance of TCP flows running in wireless networks, where it is common to have high packet losses (e.g., random losses due to wireless interference).

In Section VI we review several proposed solutions that have attracted the most research interest over the recent past. These proposals aim to solve the problem of poor utilization of high-speed and long-delay network channels by standard TCP flows. They introduce several direct and indirect approaches to more aggressive network probing. The indirect approaches combine various loss-based and delay-based ideas to create congestion control approaches that try to be aggressive enough when there are enough network resources, yet remain gentle when all resources are utilized.

Finally, we present opportunities for the future research in Section VII and conclude our survey in Section VIII.

II. Congestion Collapse

The initial TCP standard has a serious drawback: it lacks any means to adjust the transmission rate to the state of the network. When there are many users and user demands for shared network resources, the aggregate rate of all TCP senders sharing the same network can easily exceed (and in practice do exceed) the capacity of the network. It is commonly known in the flow-control world that if the offered

load in an uncontrolled distributed sharing system (e.g., road traffic) exceeds the total system capacity, the effective load will go to zero (collapses) as load increases [4] (Figure 3).

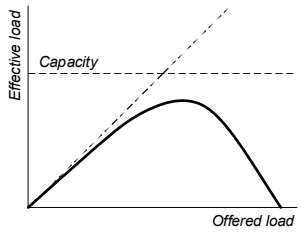


Fig. 3. Effective TCP load versus offered load from TCP senders

With regard to TCP, the origins of this effect, known as a *congestion collapse* [5], [6], [7], can be illustrated using a simple example. Let us consider a router placed somewhere between networks **A** and **B** which generate excessive amounts of TCP traffic (Figure 4). Clearly, if the path from **A** to **B** is congested by 400% (4 times more than the router can deliver), at least 75% of all packets from network **A** will be dropped and at most 25% of data packets may result in ACKs. If the reverse path from **B** to **A** is also congested (also by 400%, for example), the chance that ACK packets get through is also 25%. In other words, only 25% of 25% (i.e., 6.25%) of the data packets sent from **A** to **B** will be acknowledged successfully. If we assume that each data packet requires its own acknowledgement (not a requirement for TCP, but serves to illustrate the point), then a 75% loss in each direction causes a 93.75% drop in throughput (goodput) of the TCP-like flow. Implementing cumulative ACKs help shift the bend of the curve in Figure 3, but cumulative ACK are not able to eliminate the sharp downward bend.

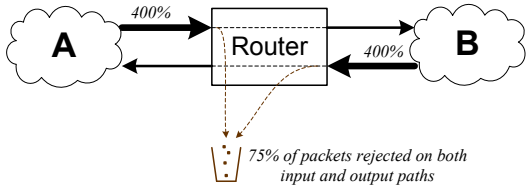


Fig. 4. Congestion collapse rationale. 75% of data packets dropped on forward path and 75% of ACKs dropped on reverse: only 6.25% of packets are acknowledged

To resolve the *congestion collapse* problem, a number of solutions have been proposed. All of them share the same idea, namely of introducing a network-aware rate limiting mechanism alongside the receiver-driven flow control. For this purpose the *congestion window* concept was introduced: a TCP sender's estimate of the number of data packets the network can accept for delivery without becoming congested. In the special case where the flow control limit (the so-called *receiver window*) is less than the congestion control limit (i.e., the *congestion window*), the former is considered a real bound for outstanding data packets. Although this is a formal definition of the real TCP rate bound, we will only consider the *congestion window* as a rate limiting factor, assuming that in most cases the processing rate of end-hosts is several orders of magnitude higher than the data transfer rate that the network can potentially offer. Additionally, we will compare different algorithms, focusing on the *congestion window* dynamics as a measure of the particular congestion control algorithm effectiveness.

In the next section we will discuss basic congestion control algorithms that have been proposed to extend the TCP specification. As we shall see, these algorithms not only preserve the idea of treating the network as a black box but also provide a good precision level to detect congestion and prevent collapse. Table I gives a summary of features of the various algorithms. Additionally, Figure 5 shows the evolutionary graph of these algorithms. However, solving the congestion problem introduces new problems that lead to network channel underutilization. Here we focus primarily on the congestion problem itself and basic approaches to improve data transfer effectiveness. In the following sections other problems and solutions will be discussed.

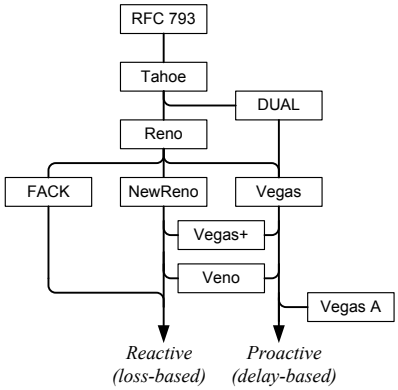


Fig. 5. Evolutionary graph of TCP variants that solve the congestion collapse problem

A. TCP Tahoe

One of the earliest host-to-host solutions to solve the congestion problem in TCP flows has been proposed by Jacobson [8]. The solution is based on the original TCP specification (RFC 793 [1]) and includes a number of algorithms that can be divided into three groups. The first group tackles the problem of an erroneous retransmission timeout estimate (RTO). If this value is overestimated, the TCP packet loss detection mechanism becomes very conservative, and performance of individual flows may severely degrade. In the opposite case, when the value of the RTO is underestimated, the error detection mechanism may perform unnecessary retransmissions, wasting shared network resources and worsening the overall congestion in the network. Since it is practically impossible to distinguish between an ACK for an original and a retransmitted packet, RTO calculation is further complicated.

The *round-trip variance estimation* ($rttvar$) algorithm tries to mitigate the overestimation problem. Instead of a linear relationship between the RTO and estimated round-trip time (RTT) value ($\beta \cdot SRTT$, in which β is a constant in range from 1.3 to 2 [1] and $SRTT$ is an exponentially averaged RTT value), the algorithm calculates an RTT variation estimate to

TABLE I
FEATURES OF TCP VARIANTS THAT SOLVE THE CONGESTION COLLAPSE PROBLEM

TCP Variant	Section	Year	Base	Added/Changed Modes or Features	Mod[1]	Status	Implementation			
							BSD[2]	Linux	Win	Mac
TCP Tahoe [8]	II-A	1988	RFC793	Slow Start, Congestion Avoidance, Fast Retransmit	S	Obsolete Standard	>4.3	1.0		
TCP-DUAL [9]	II-B	1992	Tahoe	Queuing delay as a supplemental congestion prediction parameter for Congestion Avoidance	S	Experimental				
TCP Reno [10], [11]	II-C	1990	Tahoe	Fast Recovery	S	Standard	>4.3 >F2.2	> 1.3.90	>95/NT	
TCP NewReno [12], [13]	II-D	1999	Reno	Fast Recovery resistant to multiple losses	S	Standard	>F4	> 2.1.36		>10.4.6 (opt)
TCP SACK [14]	II-E	1996	RFC793	Extended information in feedback messages	P+S+R	Standard	>S2.6, >N1.1, >F2.1R	> 2.1.90	> 98	> 10.4.6
TCP FACK [15]	II-F	1996	Reno, SACK	SACK-based loss recovery algorithm	S	Experimental	>N1.1	>2.1.92		
TCP-Vegas [16]	II-G	1995	Reno	Bottleneck buffer utilization as a primary feedback for the Congestion Avoidance and secondary for the Slow Start	S	Experimental		> 2.2.10		
TCP-Vegas+ [17]	II-H	2000	NewReno, Vegas	Reno/Vegas Congestion Avoidance mode switching based of RTT dynamics	S	Experimental				
TCP-Veno [18]	II-I	2002	NewReno, Vegas	Reno-type Congestion Avoidance and Fast Recovery increase/decrease coefficient adaptation based on bottleneck buffer state estimation	S	Experimental		> 2.6.18		
TCP-Vegas A [19]	II-J	2005	Vegas	Adaptive bottleneck buffer state aware Congestion Avoidance	S	Experimental				

[1] TCP specification modification: S = the sender reactions, R = the receiver reactions, P = the protocol specification
[2] S for Sun, F for FreeBSD, N for NetBSD

establish a fine-grained upper bound for the RTO ($SRTT + 4 \cdot rttvar$).

The *exponential retransmit timer backoff* algorithm solves the underestimation problem by doubling the RTO value on each retransmission event. In other words, during severe congestion, detection of subsequent packet losses results in exponential RTO growth, significantly reducing the total number of retransmissions and helping stabilize the network state.

The ACK ambiguity problem is resolved by *Karn's clamped retransmit backoff* algorithm [20]. Importantly, the RTT of a data packet that has been retransmitted is not used in calculation for the average RTT and RTT variance, and thus it has no impact on the RTO estimate.

The second group of algorithms enhances the detection of packet losses. The original TCP specification defines the RTO as the only loss detection mechanism. Although it is sufficient to reliably detect all losses, this detection is not fast enough. Clearly, the minimum time when loss can be detected is the RTT—i.e., if the receiver is able to instantly detect and report a loss to the sender, the report will reach the sender exactly one RTT after sending the lost packet. The RTO, by definition, is greater than RTT. If we require that TCP receivers immediately reply to all out-of-order data packets with reports of the last in-order packet (a duplicate ACK) [21], the loss can be detected by the *Fast Retransmit* algorithm [22], almost within the RTT interval. In other words, assuming the probability of packet reordering and duplication in the network is negligible, the duplicate ACKs can be considered a reliable loss indicator. Having this new indicator, the sender can retransmit lost data without waiting for the corresponding RTO event.

The third and most important group includes the *Slow Start* and *Congestion Avoidance* algorithms. These provide two slightly different distributed host-to-host mechanisms which allow a TCP sender to detect available network resources and adjust the transmission rate of the TCP flow to the detected limits. Assuming the probability of random packet corruption during transmission is negligible ($\ll 1\%$), the sender can treat all detected packet losses as congestion indicators. In addition, the reception of any ACK packet is an indication that the network can accept and deliver at least one new packet (i.e., the ACKed packet has left and a new one can enter the network). Thus the sender, reasonably sure it will not cause congestion, can send at least the amount of data that has just been acknowledged. This in-out packet balancing is called the *packet conservation principle* and is a core element, both of *Slow Start* and of *Congestion Avoidance*.

In the *Slow Start* algorithm, reception of an ACK packet is considered an invitation to send double the amount of data that has been acknowledged by the ACK packet (*multiplicative increase* policy). In other words, instead of a step-like growth (Figure 6) in the number of *outstanding* packets (as given in the original specification [1]), this growth follows an exponential function on an RTT-defined scale (Figure 7). The word "slow" in the algorithm name makes reference to this difference. If a packet loss is detected (i.e., the network is experiencing congestion because all network resources have been utilized), the congestion window is reset to the initial value (e.g., one) to ensure release of network resources. Graphs on Figure 7 show two cases of the *congestion window* dynamics: the left graph represents the case when the receiver cannot process at the receiving rate (i.e., the original assumption of TCP), and the right graph shows the congestion window

dynamics when the network cannot deliver everything at the transmitted rate.

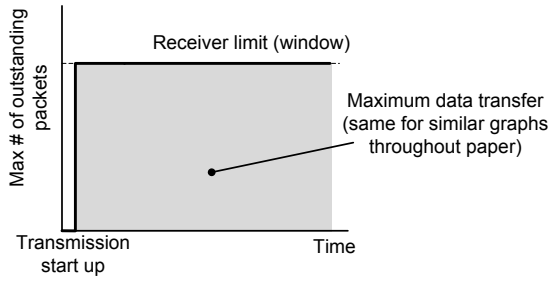


Fig. 6. Outstanding data packets allowance dynamics as defined in RFC793 (network limits are not considered)

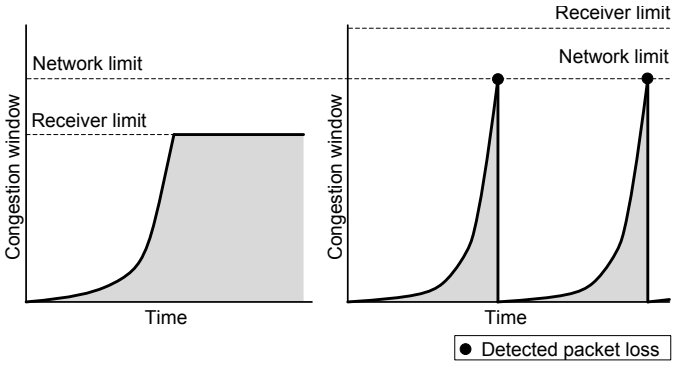


Fig. 7. Congestion window dynamics and effectiveness of *Slow-Start* if limit is imposed by legacy flow control (left) and network (right)

We can define algorithm effectiveness as the ratio of the area below the congestion window graph (e.g., Figure 7, hatched area) to the area below the limit line (Figure 7, under "Network Limit" line). It is clear (observing the right graph in Figure 7) that where the available network resources are lower than limits imposed by the receiver, the effectiveness, in the long term, of the *Slow Start* algorithm is very low.

The other algorithm of the third group is *Congestion Avoidance*. It is aimed at improving TCP effectiveness in networks with limited resources, i.e., where the network is a real transmission bottleneck. In comparison to the *Slow Start*, this algorithm is much more conservative in response to received ACK packets and to detection of packet losses. As opposed to doubling, the *congestion window* increases by one only if all data packets have been successfully delivered during the last RTT (*additive increase* policy). And in contrast to restarting at one after a loss, the congestion window is merely halved (*multiplicative decrease* policy). Jacobson's analysis [8] has shown that to achieve network decongestion, exponentially reducing network resource utilization by each individual flow is sufficient. The *multiplicative decrease* policy mimics such exponential behavior when several packets in succession are determined as lost (e.g., during the persistent congestion state). As can be seen in Figure 8, the *Congestion Avoidance* algorithm is quite effective in the long term. The tradeoff is a slow discovery of available network resources due to the conservative rate of the additive phase.

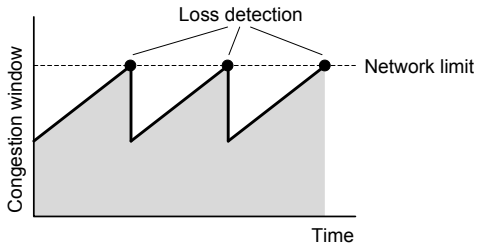


Fig. 8. Congestion window dynamics and effectiveness of *Congestion Avoidance*

The implementation of TCP Tahoe includes both *Slow Start* and *Congestion Avoidance* algorithms as distinct operational phases. This combines fast network resource discovery and long-term efficiency. For phase-switching purposes, a threshold parameter ($ssthresh$) is introduced. This threshold determines the maximum size of the *congestion window* in the *Slow Start* phase, and any detected packet loss adjusts the threshold to half of the current *congestion window*. The *congestion window* itself, as in the *Slow Start* algorithm, is always reset to a minimum value upon loss detection. As long as the value of the *congestion window* is lower than the threshold parameter, the *Slow Start* phase is used. Once the window is greater than the threshold, *Congestion Avoidance* is used. This is known as the *Slow Start-Congestion Avoidance* phase cycle (Figure 9).

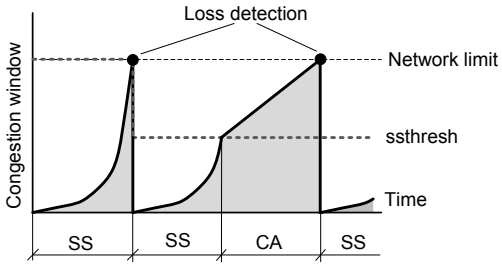


Fig. 9. Congestion window dynamics of combined *Slow-Start* (SS) *Congestion Avoidance* (CA)

Effectiveness is not the only important parameter of congestion control algorithms. Due to the resource-sharing nature of IP networks, TCP algorithms should enforce fair resource sharing. Chiu and Jain [23] developed a fairness measure F (the so-called *Jain's fairness index*) as a function of network resources consumed by each user sharing the same path:

$$F = \frac{(\sum_{i}^{n} f_i)^2}{n \cdot \sum_{i}^{n} f_i^2}$$

where n is the number of users sharing the path and f_i is the network share of i^{th} user. If we assume that each user has only one TCP connection per particular network path, then *Jain's index* can be considered a fairness measure for TCP flows. This index ranges from 0 to 1, where 1 is achieved if and only if each flow has an equal (fair) share ($f_i = f_j \ \forall i, j$)

and tends to zero if a single flow usurps all network resources ($lim_{n\to\infty} F = 0$).

Slow Start and *Congestion Avoidance* exhibit good fairness ($F \to 1$) under certain network conditions as follows. Let us consider two flows competing with each other on the same network path and with no other flows present. If we assume that (a) the network share for each flow is directly proportional to its congestion window size, (b) both flows have equal RTT values, and (c) we can simultaneously detect packet losses (a so-called *synchronized loss* environment), then the network share dynamics for each algorithm can be represented by the convergence diagrams in Figures II-A and II-A. The *equal share* line represents states when network resources are fairly distributed between flows and the *network limit* line when all network resources are consumed (either by one or both flows). These diagrams show how network resource proportions would change (paths $x_0 - x_1$, $x_1 - x_2$, $\ldots$, $x_{n-1} - x_n$) if two TCP flows started competing with each other from an initial state x_0 under the ideal network conditions.

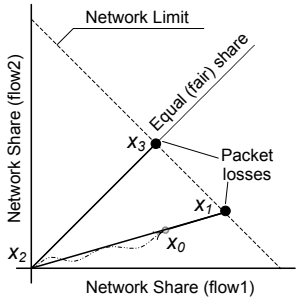


Fig. 10. Slow-Start Algorithm
$x_0 - x_1, \ldots, x_n - x_{n+1}$ multiplicative increase (both flows have the same increase rate of their congestion windows)
$x_1 - x_2$ equalization of the congestion window sizes

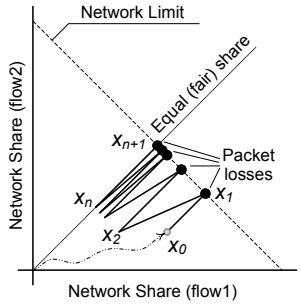


Fig. 11. Congestion avoidance (AIMD)
$x_0 - x_1, \ldots, x_n - x_{n+1}$ additive increase (both flows have the same increase rate of their congestion windows)
$x_1 - x_2, \ldots, x_{n-1} - x_n$ multiplicative decrease (a flow with the larger congestion window decreases more than a flow with the smaller)

In Figure II-A the aggressive (multiplicative) congestion window increase in *Slow Start* favors the flow having a larger network share. More precisely, the slope of $x_0 - x_1$ segment is proportional to the ratio between each flow's share in state x_0. After detection of a packet loss, both flows reset their congestion windows ($x_1 - x_2$). Obviously, reseting of the congestion window equalizes the network share of the flows that provides fairness of the network resource distribution in the future (the flows become locked-in between the states x_2 and x_3).

Congestion Avoidance ensures a uniform congestion window increase by each flow from any initial state (45° slope of $x_1 - x_2$ and $x_n - x_{n+1}$ segments in Figure II-A). This property eliminates the necessity of the congestion window equalization. Instead, to provide fair network usage between flows, it is enough that the flow having a larger network share decreases by a greater amount. In fact, the multiplicative decrease (i.e., congestion window halving) as a reaction to a packet loss in *Congestion Avoidance* guarantees share equalization (fairness) in a finite number of steps.

A convergence diagram of TCP Tahoe can be represented as a combination of the *Slow Start* and *Congestion Avoidance* diagrams. Depending on the values of the *Slow Start* thresholds, the initial dynamics can follow the *Slow Start* path ($x_0 - x_1$ in Figure II-A) or the *Congestion Avoidance* path ($x_0 - x_1$ in Figure II-A), or be a combination of both algorithms. Because the reaction to a packet loss in TCP Tahoe is the same as in *Slow Start* (i.e., congestion window reset), exactly one loss event is enough to equalize shares (similar to path $x_1 - x_2$ in Figure II-A).

B. TCP DUAL

TCP Tahoe (Section II-A) has rendered a great service to the Internet community by solving the congestion collapse problem. However, this solution has an unpleasant drawback of straining the network with high-amplitude periodic phases. This behavior induces significant periodic changes in sending rate, round-trip time, and network buffer utilization, leading to variability in packet losses.

Wang and Crowcroft [9] presented TCP DUAL, which refines the *Congestion Avoidance* algorithm. DUAL tries to mitigate the oscillatory patterns in network dynamics by using a proactive congestion detection mechanism coupled with softer reactions to detected events. More specifically, it introduces the *queuing delay* as a prediction parameter of the network congestion state.

Let us assume that routes do not change during the transmission and that the receiver acknowledges each data packet immediately. Then we can consider the minimal RTT value observed by the sender (RTT_{min}) as a good indication that the path is in a congestion-free state (left diagram in Figure 12). If we make one more assumption that an increase of the RTT can only occur due to increasing buffer utilization, the difference between the measured and the minimal RTT value (queuing delay $Q = RTT - RTT_{min}$) can be viewed as an indicator of the congestion level in the path (right diagram in Figure 12).

To quantify the congestion level, DUAL additionally maintains a maximum RTT value observed during the transmission (RTT_{max}). The difference between maximum and minimum RTT values is considered a measure of the maximum congestion level (i.e., the maximum *queuing delay* $Q_{max} = RTT_{max} - RTT_{min}$). Finally, a fraction of the maximum queuing delay ($Q_{thresh} = \alpha \cdot Q_{max}$, where $0 < \alpha < 1$) serves as a threshold, which, when exceeded, indicates the congested network state.

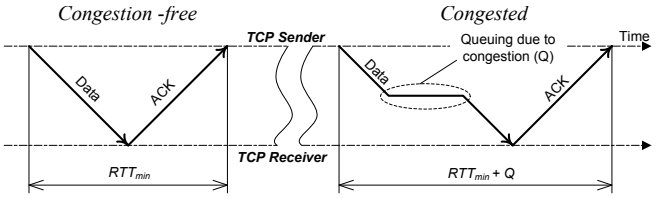


Fig. 12. Correlation between RTT dynamics and congestion situation

In the proposal [9], the delay threshold in DUAL is selected as half the maximum queuing delay ($Q_{thresh} = Q_{max}/2$), and the congestion estimation is performed once per RTT period based on the average RTT value ($Q = RTT_{avg} - RTT_{min}$). If this threshold is exceeded ($Q > Q_{thresh}$), the congestion window decreases by $1/8^{th}$ (i.e., applied *multiplicative decrease* policy). As we can see from theoretical congestion window dynamics of TCP DUAL (Figure 13), the effectiveness is greatly improved compared to Tahoe (i.e., graphically, the hatched area is proportionately larger). However, there are a number of trade-offs.

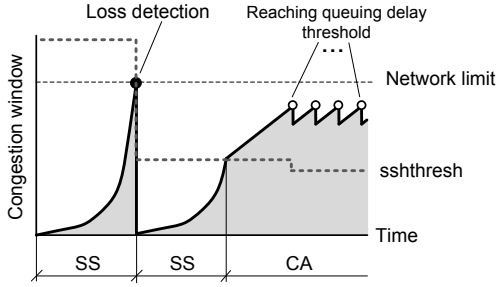


Fig. 13. Congestion window dynamics of TCP DUAL
(SS: the *Slow Start* phase, CA: the *Congestion Avoidance* phase)

If the network saturation point is estimated incorrectly, the flow cannot utilize the available network resources fairly and effectively. On the one side, in the case where the threshold is underestimated (e.g., observed RTT_{max} is not the real maximum) network resources will be underutilized. On the other side, threshold overestimation can potentially cause an unfair resource distribution between different TCP DUAL flows. For example, if a DUAL flow is already transmitting data when a new DUAL flow appears, the new flow will observe a higher RTT_{min} value and overestimate the queuing delay threshold. The flow with the lower queuing threshold (the old flow) has a higher probability of predicting the congestion state and trigger congestion window reduction, while the other flow will continue congestion window growth without noticing anything abnormal. Thus, the new flow can potentially capture a larger share of the network resources.

C. TCP Reno

Reducing the congestion window to zero as a reaction to packet loss, as occurs with TCP Tahoe (Section II-A), is rather draconian and can, in some cases, lead to significant throughput degradation. For example, a 1% packet loss rate can cause up to a 75% throughput degradation of a TCP flow running the Tahoe algorithm [10]. To resolve this problem, Jacobson [10] revised the original composition of *Slow Start* and *Congestion Avoidance* by introducing the concept of differentiating between major and minor congestion events.

A loss detection through the retransmission timeout indicates that for a certain time interval (as an example, RTO minus RTT) some major congestion event has prevented delivery of any data packets on the network. Therefore, the sender should apply the conservative policy of resetting the congestion window to a minimal value.

Quite a different state can be inferred from a loss detected by duplicate ACKs. Suppose the sender has received four ACKs, where the first one acknowledges some new data and the rest are the exact copies of the first one (usually referred to as three duplicate ACKs). The duplicate ACKs indicate that the successive packets of the sequence have failed to arrive. Nonetheless, each ACK—including the duplicates—indicates the successful delivery of a data packet. The sender, in addition to detecting the lost packet, is also observing the ability of the network to deliver some data. Thus, the network state can be considered to be lightly congested, and the reaction to the loss event can be more optimistic. In TCP Reno, the optimistic reaction is to use the *Fast Recovery* algorithm [11].

The intention of *Fast Recovery* is to halve a flow's network share (i.e., to halve the congestion window) and to taper back network resource probing (holding all growth in the congestion window) until the error is recovered. In other words, the sender stays in *Fast Recovery* until it receives a non-duplicate acknowledgment. The algorithm phases are illustrated in Figure 14, where congestion window sizes (*cwnd*) in various states are denoted as the line segments above the *State* lines, and the arrows indicate the effective congestion window size—the amount of packets in transit.

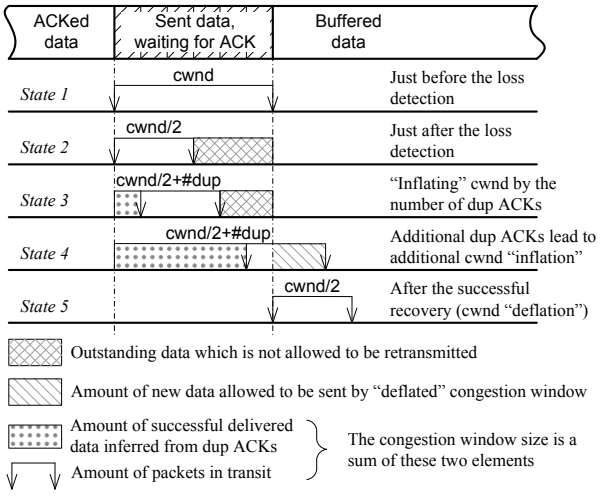


Fig. 14. Characteristic states of TCP Reno's *Fast Recovery*

The transition from *State 1* to *State 2* shows the core concept of optimistic network share reduction, using the *multiplicative decrease* policy. After the reduction (i.e., from $cwnd$ to $cwnd/2$), the algorithm not only retransmits the oldest unacknowledged data packet (i.e., applies the *Fast Retransmit* algorithm), but also *inflates* the congestion window by the number of duplicate packets (see transition from *State 2* to

State 3 in Figure 14). As we already know, an ACK indicates delivery of at least one data packet. Thus, if we want to maintain a constant number of packets in transit, we have to inflate our congestion window to open a slot for sending new data (*State 4* in Figure 14). Without this increase, new packets cannot be sent before the error is recovered, and the amount of packets in transit can decrease more than expected.

In the final stage (*State 5*), when a non-duplicate ACK is received, we want to resume *Congestion Avoidance* with half of the original congestion window. With high probability, the non-duplicate ACK will acknowledge delivery of all data packets previously inferred by the duplicate ACKs previously received. At this point, *congestion window* deflation to $cwnd/2$ (to the value just after entering recovery, *State 2* in Figure 14) is a simple and reliable way to ensure the target exit state from *Fast Recovery*.

The resulting theoretical congestion window dynamics in TCP Reno are presented in Figure 15. Compared to the dynamics of TCP Tahoe (Figure 9), the overall effectiveness in the steady state is considerably improved by replacing *Slow Start* phases after each loss detection by typically shorter *Fast Retransmit* phases.

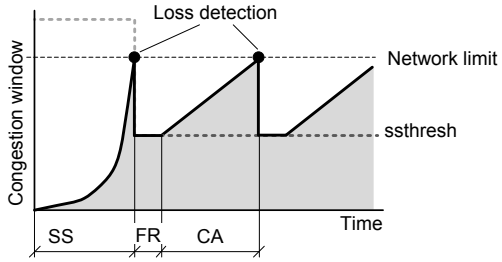


Fig. 15. Congestion window dynamics of TCP Reno
(SS: the *Slow Start* phase, CA: the *Congestion Avoidance* phase, FR: the *Fast Recovery* phase)

In fact, recovering from a single loss would usually occur within one RTT. However, efficiency is improved not only by shortening the recovery period, but also by allowing data transfers during the recovery. Having substantial performance improvement compared to Tahoe, TCP Reno remains fair to other TCP Reno flows (in terms defined in Section II-A). If we try to build a convergence diagram, it would match the diagram for the *Congestion Avoidance* algorithm in Figure II-A exactly. However, a slightly worse situation can be observed when a TCP Reno flow competes with a Tahoe flow. Unequal reactions to packet loss detection lead to shifting the distribution of network resources to the Reno side. This can be demonstrated using the convergence diagram in Figure 16. With a finite number of steps, the system reaches a steady state in which the Reno flow has a larger share of network resources. To quantify fairness in this case, one can easily calculate the *Jain's fairness index* (see Section II-A). In Figure 16, this value equals 0.9 (after the convergence—state x_{n+1}—network shares are distributed as 2:1 in favor of a Reno flow). This can be considered an acceptable level for the transition period when the congestion control algorithm is changed from Tahoe to Reno at all network hosts.

A comparison to TCP DUAL shows that, in an ideal network

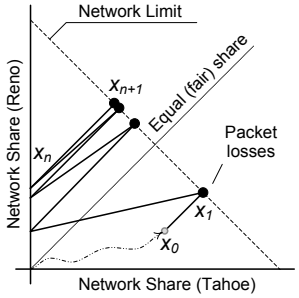


Fig. 16. Convergence diagram when Reno flow is competing with Tahoe flow
$x_0 - x_1, \ldots, x_n - x_{n+1}$ additive increase (both flows have the same increase rate of their congestion windows)
$x_1 - x_2, \ldots, x_{n-1} - x_n$ Tahoe flow reset its congestion window but Reno flow only halves it

environment with only one TCP flow present, the DUAL algorithm will normally outperform Reno. But DUAL has several important drawbacks. First, the delay characteristic is not always a true congestion indicator, which can lead to network resource underutilization or unfair distribution of network resources. Second, there is an open question about how well the DUAL algorithm performs in less ideal environments and when DUAL flows compete with other DUAL or Tahoe flows. Clearly, in a situation of higher packet losses, the performance of DUAL and Tahoe would be about the same, and as a result, Reno could outperform the both of them. Additionally, doubts about DUAL's fairness to other DUAL flows, as discussed in Section II-B, tend to give further favor to TCP Reno.

Because of its simplicity and performance characteristics, Reno is generally the congestion control standard for TCP. At the same time, there are a wide range of network environments where Reno has inadequate performance. For example, it has severe performance degradation in the presence of consecutive packet losses, random packet losses, and reordering of packets. It is also unfair if competing flows have different RTTs, and it does not utilize high-speed/long-delay network channels efficiently.

In the remainder of this paper we will discuss a number of the most important TCP proposals which address these issues without deviating from the original host-to-host principle of TCP.

D. TCP NewReno

One of the vulnerabilities of TCP Reno's *Fast Recovery* algorithm manifests itself when multiple packet losses occur as part of a single congestion event. This significantly decreases Reno's performance in heavy load environments. This problem is demonstrated in Figure 17, where a single congestion event (e.g., a short burst of cross traffic) causes the loss of several data packets (indicated by **x**). As we can see, the desired optimistic reaction of *Fast Recovery* (i.e., the congestion window halving) suddenly transforms into a conservative exponential congestion window decrease. That is, the first loss causes entry into the recovery phase and the halving of the congestion window. The reception of any non-duplicate ACK would finish

the recovery. However, the subsequent loss detections cause the congestion window to decrease further, using the same mechanisms of entering and exiting the recovery state.

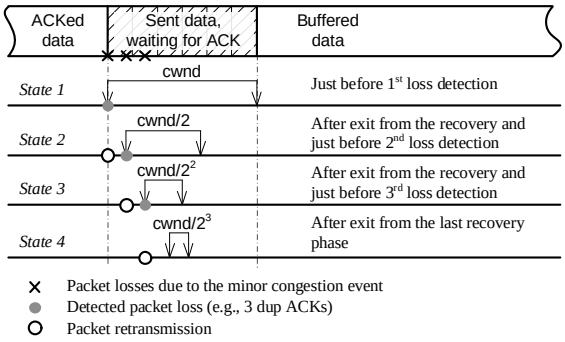


Fig. 17. The performance problem in Reno's *Fast Recovery*

In one sense, this exponential reaction to multiple losses is expected from the congestion control algorithm, the purpose of which is to reduce consumption of network resources in complex congestion situations. But this expectation rests on the assumption that congestion states, as deduced from each detected loss, are independent, and in the example above this does not hold true. All packet losses from the original data bundle (i.e., from those data packets outstanding at the moment of loss detection) have a high probability of being caused by a single congestion event. Thus, the second and third losses from the example above should be treated only as requests to retransmit data and not as congestion indicators. Moreover, reducing the congestion window does not guarantee the instant release of network resources. All packets sent before the congestion window reduction are still in transit. Before the new congestion window size becomes effective, we should not apply any additional rate reduction policies. This can be interpreted as reducing the congestion window no more often than once per one-way propagation delay or approximately $RTT/2$.

Floyd et al. [12], [13] introduce a simple refinement of Reno's *Fast Recovery*. It solves the ambiguity of congestion events by restricting the exit from the recovery phase until all data packets from the initial congestion window are acknowledged. More formally, the NewReno algorithm adds a special state variable to remember the sequence number of the last data packet sent before entering the *Fast Recovery* state. This value helps to distinguish between *partial* and *new data* ACKs. The reception of a *new data ACK* means that all packets sent before the error detection were successfully delivered and any new loss would reflect a new congestion event. A *partial ACK* confirms the recovery from only the first error and indicates more losses in the original bundle of packets.

Figure 18 illustrates the differences between Reno and NewReno. Similar to the original Reno algorithm, reception of any duplicate ACKs triggers only the *inflation* of the congestion window (*States 3*, *4*, *6*). A *partial ACK* provides the exact information about some part of the delivered data. Therefore, reaction to partial ACK is only a *deflation* of the congestion window (*State 4*) and a retransmission of the next unacknowledged data packet (*State 5*). Finally, exit from the NewReno's *Fast Recovery* can proceed only when the sender receives a *new data* ACK, which is accompanied by the full congestion window deflation (*State 7* in Figure 18).

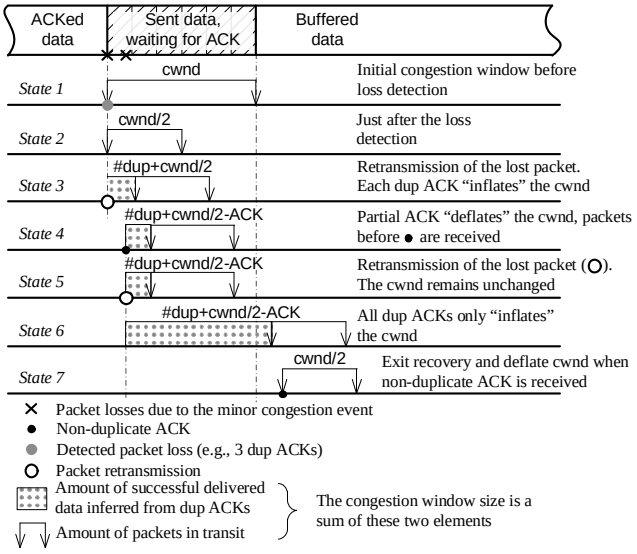


Fig. 18. Characteristic states of TCP NewReno's *Fast Recovery*

Notice that during the recovery phase, duplicate acknowledgments transfer their role as error indicators to the partial ACKs. Retransmission of the first lost packet and reception of the corresponding ACK will take exactly one RTT, and the sender therefore can be absolutely sure that during this interval all previously sent data will be either delivered or lost. That is, this data no longer consumes the network resources. Partial ACKs can be sent by the receiver only if more than one packet is lost from the original packet bundle. Thus there is no reason for the sender to wait for additional signals before retransmitting the lost packet inferred from the partial ACK.

NewReno modifies only the *Fast Recovery* algorithm by improving its response in the event of multiple losses. Meanwhile, in the steady state performance and fairness characteristics are similar to the ones shown in Section II-C. A slightly more aggressive recovery procedure would allow a NewReno flow in some cases to obtain more network resources than a competing Reno flow. But generally, this imbalance only happens due to the inability of Reno itself to utilize the network resources under those network conditions effectively. For this reason, we consider NewReno to have the same fairness characteristics as Reno.

E. TCP SACK

The problem with Reno's *Fast Recovery* algorithm discussed in the Section II-D arises solely because the receiver can report limited information to the sender. The TCP specification [1] defines that the only feedback message be in the form of *cumulative ACKs*, i.e., acknowledgments of only the last in-order delivered data packet. This property limits the ability of the sender to detect more than one packet loss per RTT. For example, if a second and a third data packet from some continuous TCP stream are lost, the receiver, according to the *cumulative ACK* policy, would reply to fourth and consecutive

packets with duplicate acknowledgments of the first packet (Figure 19). Clearly, the loss can be detected no sooner than after one RTT. Moreover, because *Fast Recovery* assumes loss of only one data packet—i.e., only one packet will be retransmitted after a loss detection—loss of the third data packet will be detected after another RTT at best. Thus, a duration of the recovery in Reno is directly proportional to the number of packet losses and RTT.

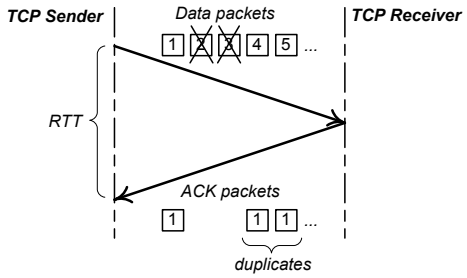


Fig. 19. Duplicate ACKs allow loss detection no sooner than after one RTT

NewReno resolves Reno's problem of excessive rate reducing in the presence of multiple losses, but it does not solve the fundamental problem of prolonged recovery. The recovery process can be sped up if the sender retransmits several packets instead of a single one upon error detection. However, this technique assumes certain patterns of packet losses and may just waste network resources if actual losses deviate from these patterns.

If a receiver can provide information about several packet losses within a single feedback message, the sender would be able to implement a simple algorithm to resolve the long recovery problem. Moreover, Reno's problem discussed in Section II-D can be solved by restricting the congestion window reduction to no more than once per RTT period, instead of implementing the NewReno algorithm. The rationale behind this solution is that in the worst case, the interval between the first and last data packets sent before reception of any ACK is exactly one RTT (Figure 20). All loses, if any, will be reported to the sender within the next RTT. Thus, if we apply the mechanism of limiting the congestion window reduction to no more than once per RTT period to the problem illustrated in Figure 17, the first error detection should cause retransmission and shrink the congestion window. The rest of the losses in the original packet bundle would be reported within one RTT, and thus will cause only retransmission of the lost packets, preserving the value of the congestion window.

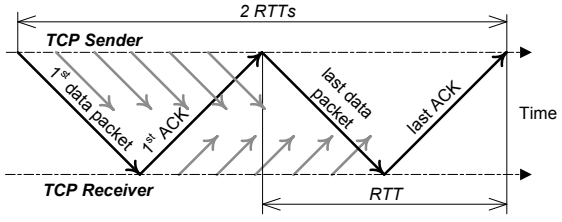


Fig. 20. The interval between the first and last data packets sent before reception of any ACK is 2×RTT in the worst case

Mathis et al. [14] address the problem of limited information available in a cumulative ACK. As a solution, they propose extending the TCP protocol by standardizing the selective acknowledgment (SACK) option. This option provides the ability for the receiver to report blocks of successfully delivered data packets. Using this information, TCP senders can easily calculate blocks of lost packets (*gaps* in sequence numbers) and quickly retransmit them (Figure 21).

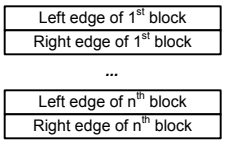


Fig. 21. SACK option
Left edge – the first sequence number of the block,
Right edge – the sequence number immediately following the last sequence number of the block

Unfortunately the SACK mechanism has serious limitations in its current form. The TCP specification restricts the length of the option field to 40 bytes. A simple calculation reveals that the SACK option can contain at most four blocks of data packets received in order (2 bytes to identify option and specify option length, and up to four pairs of 4-byte sequence numbers [14]). The situation becomes aggravated if we want to use additional TCP options, which decrease the space for the sequence number pairs being included in SACK. For example, the *Timestamp* option [24] reduces the available space in the TCP header by 8 bytes, which decreases SACK space to only 3 *gaps* of lost packets. In some environments, the pattern of packet losses may easily exceed this SACK limit. In the worst case, when every other packet is lost, this limit is exceeded just after the first 4 packets are received (thus, 4 packets being lost). The inability of the receiver to quickly indicate all detected losses returns us to the original problem. Although this worst-case situation is unlikely to happen in wired networks (since during congestion events consecutive packets are usually dropped), random losses in wireless networks can show patterns approximating the worst case. This observation shows that SACK is not a universal solution to the multiple loss problem.

F. TCP FACK

Although SACK (Section II-E) provides the receiver with extended reporting capabilities, it does not define any particular congestion control algorithms. We have informally discussed one possible extension of the Reno algorithm utilizing SACK information, whereby the congestion window is not multiplicatively reduced more than once per RTT. Another approach is the FACK (Forward Acknowledgments) congestion control algorithm [15]. It defines recovery procedures which, unlike the *Fast Recovery* algorithm of standard TCP (TCP Reno), use additional information available in SACK to handle error recovery (flow control) and the number of outstanding packets (rate control) in two separate mechanisms.

The flow control part of the FACK algorithm uses selective ACKs to indicate losses. It provides a means for timely retransmission of lost data packets, as well. Because retransmitted

data packets are reported as lost for at least one RTT and a loss cannot be instantly recovered, the FACK sender is required to retain information about retransmitted data. This information should at least include the time of the last retransmission in order to detect a loss using the legacy timeout method (RTO).

The rate control part, unlike Reno's and NewReno's *Fast Recovery* algorithms, has a direct means to calculate the number of outstanding data packets using information extracted from SACKs. Instead of the congestion window inflation technique, the FACK maintains three special state variables (Figure 22): (1) H, the highest sequence number of all sent data packets—all data packets with sequence number less than H have been sent at least once; (2) F, the forward-most sequence number of all acknowledged data packets—no data packets with sequence number above F have been delivered (acknowledged); and (3) R, the number of retransmitted packets.

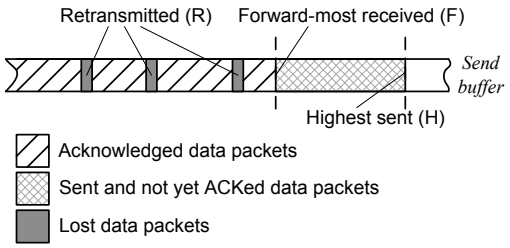


Fig. 22. Special state variables in the FACK algorithm

The simple relation $H - F + R$ provides a reliable estimate (in the sense of robustness to ACK losses) of outstanding data packets in the network. This estimate can be utilized by the sender to decide whether or not to send a new portion of data. More formally, data can be sent when the calculated number of outstanding data packets is under an allowed limit (the congestion window).

Simulation results [15] confirm that FACK has a much faster recovery time from errors than Reno or NewReno. In fact, advantages of FACK have long been widely recognized and FACK has been an embedded part of the Linux kernel since the 2.1.92 version. Because FACK modifies only reactions in the recovery phase, the steady-state characteristics of effectiveness and fairness are exactly the same as for Reno (see Section II-C).

G. TCP Vegas

The approaches discussed in Sections II-C, II-D, and II-F improve various aspects of Tahoe, Reno and NewReno congestion controls. However, all of them share the same reactive method of rate adaptation. That is, each of them detects that the network is congested only if some packets are lost. Moreover, these variants of TCP bring about packet losses because their algorithms can grow packet transmission rates to the point of network congestion. Therefore, the problem discussed in Section II-B (i.e., induced periodic changes in sending rate, round-trip time, network buffer utilization, packet losses, etc.), also applies to Reno, NewReno, and FACK congestion control algorithms.

TCP DUAL (Section II-B) makes an attempt to provide a proactive method of quantifying the congestion level before an actual congestion event occurs using an estimate of queuing delay. But the solution only mitigates the oscillatory patterns of network parameters (RTT, buffer utilization, etc.) and never fully eliminates them. Moreover, as mentioned in Section II-B, fairness of the DUAL algorithm is questionable.

Brakmo and Peterson [16] proposed the Vegas algorithm as another proactive method to replace the reactive *Congestion Avoidance* algorithm. The key component is making an estimate of the used buffer size at the bottleneck router. Similar to the DUAL algorithm, this estimate is based on RTT measurements. The minimal RTT value observed during the connection lifetime is considered a baseline measurement indicating a congestion-free network state (analogous to Figure 12). In other words, a larger RTT is due to increased queuing in the transmission path. Unlike DUAL, Vegas tries to quantify, not a relative, but an absolute number of packets enqueued at the bottleneck router as a function of the expected and actual transmission rate (Figure 23).

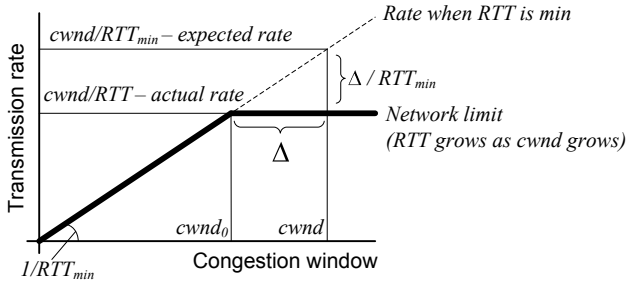


Fig. 23. TCP Vegas—the utilized buffer size Δ as function of expected and actual rate

The expected rate (dashed line in Figure 23) is a theoretical rate of a TCP flow in a congestion-free network state. This rate can occur if all transmitted data packets are successfully acknowledged within the minimal RTT (i.e., no loss, no congestion). Assuming that RTT_{min} is constant, the expected rate is directly proportional to the size of the congestion window with a proportionality coefficient of $1/RTT_{min}$.

The actual rate (bold solid line in Figure 23) can be expressed as the ratio between the current congestion window and the current RTT value. However, due to the finite capacity of the path, we can always find a point $cwnd_0$ on the graph when the actual rate is numerically equal to the expected rate, and all attempts to send at a faster rates (i.e., $> cwnd_0/RTT_{min}$) will fail. Clearly, the number of packets enqueued during the last RTT is the difference Δ between the current congestion window and the inflection point in our graph; thus we have $\Delta = cwnd - cwnd_0$. According to our assumptions, this excess of data packets is the only cause of a corresponding RTT increase. Thus, Δ can be expressed as a function of the congestion window size, RTT and RTT_{min}:

$$\Delta = cwnd \times \frac{RTT - RTT_{min}}{RTT}$$

Vegas incorporates this Δ measure into the *Congestion Avoidance* phase to control the sender's window of allowed

outstanding data packets (see beneath "Congestion Avoidance," Figure 24). In other words, once every RTT Vegas checks the difference Δ between the expected rate (small circles in Figure 24) and the actual rate (solid line in Figure 24). If Δ is more than the predefined threshold β (e.g., according to Linux implementation, more than 4), the congestion window is decreased by one; otherwise, it is increased by one. However, to mitigate the effects of network parameter fluctuations and to provide system stabilization, the proposed algorithm defines a control *dead-zone* (hatched area in Figure 24) using additional threshold α. That is, the congestion window increase is allowed only if Δ is strictly less than α (e.g., less than 2). If Δ is between α and β, the system is considered to be in a steady state and no modifications to the congestion window are applied.

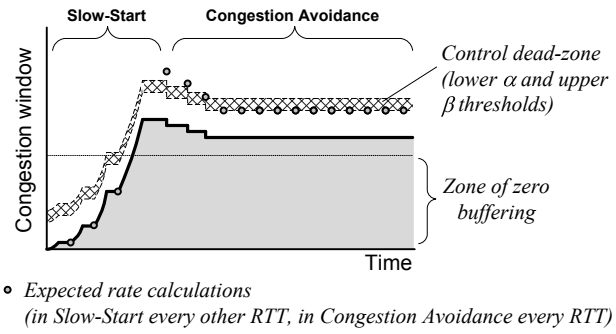


Fig. 24. TCP Vegas – congestion window dynamics and corresponding estimates of bottleneck buffer size Δ

If no packets are dropped in the network, Vegas controls the congestion window using an additive increase and additive decrease (AIAD) policy. Reactions to packet losses are defined by any of the standard congestion control algorithms (either Reno, NewReno, or FACK).

Additionally, Vegas revises the *Slow Start* algorithm by slowing-down the opportunistic network resource probing. In particular, the updated algorithm restricts the congestion window to increase every other RTT (see beneath "Slow-Start," Figure 24). This period is required in order to employ the bottleneck buffer estimation technique. As soon as Vegas detects increasing queues in bottleneck routers (i.e., Δ becomes larger than α), the *Slow Start* algorithm terminates and transfers control to the Vegas *Congestion Avoidance* algorithm.

Although the *Slow Start* modification was designed to reduce network stress, experimental results [16] show almost no measurable impact. The main reason for this is the negligible working time of the *Slow Start* phase compared to the *Congestion Avoidance* phase. In practice, available Linux implementations do not perform any changes to the original *Slow Start* algorithm and implement only the modified *Congestion Avoidance* phase.

As we can see from Figure 24, TCP Vegas has the amazing property of rate stabilization in a steady state, which can significantly improve the overall throughput of a TCP flow. Unfortunately, despite this and other advantages, later research [17], [25], [19] discovered a number of issues, including the inability of Vegas to get a fair share when competing with aggressive TCP Reno-style flows (a reactive approach is always more aggressive). It also underestimates available network resources in some environments (e.g., in the case of multipath routing) and has a bias to new streams (i.e., newcomers get a bigger share) due to inaccurate RTT_{min} estimates.

H. TCP Vegas+

Hasegawa et al. [17] have recognized a serious problem in TCP Vegas which prevents any attempts to deploy it. The Vegas proactive congestion-prevention mechanism (limiting buffering in the path) cannot effectively compete with the highly deployed Reno reactive mechanism (inducing network buffering and buffer overflowing). This point can be illustrated using an idealized convergence diagram for competition between Reno and Vegas flows (Figure 25). While there is no buffering on the path, both flows slowly increase their share of network resources (x_0–x_1). Excessive buffering forces Vegas to decrease its congestion window, but the Reno flow, unaware of this buffering, continues acquiring more network resources (x_1–x_2). That is, opposite reactions of the two different congestion control algorithms (one growing, one diminishing) maintain the fixed buffering level in the network, leading to the proactive algorithm being completely pinched off (x_2). Buffers are purged only when the Reno flow detects a packet loss (x_3). After that, the convergence dynamics will start looping along the path x_4–x_5–x_2–x_3–x_4.

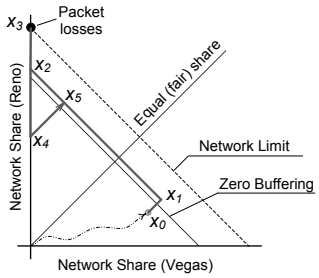


Fig. 25. Convergence diagram when an ideal Vegas flow is competing with a Reno flow

TCP Vegas+ was proposed as a way to provide a way of incremental Vegas deployment. For this purpose, Vegas+ borrows from both the reactive (Reno-like *aggressive*) and proactive (Vegas-like *moderate*) congestion avoidance approaches. More specifically, the *Congestion Avoidance* phase of Vegas+ initially assumes a Vegas-friendly network environment and employs bottleneck buffer estimation to control the congestion window (i.e., Vegas rules). At the moment when an internal heuristic detects a Vegas-unfriendly environment, *Congestion Avoidance* falls back to the Reno algorithm.

The Vegas-friendliness/unfriendliness detection heuristic is based on a trend estimate of the RTT. The special state variable C is increased if the sender estimates an increase in the RTT and concurently the size of the congestion window is unchanged or even reduced. In the opposite case, if the estimated RTT grows smaller, C is decreased. Clearly, large values of C indicate a Vegas-unfriendly network state (i.e., if the congestion window is stable, the RTT also should be stable). Transition to the *aggressive* mode is triggered when

C exceeds a predefined threshold. Return to the *moderate* mode occurs only when C becomes zero. Vegas+ additionally defines two special cases for modifying the state variable: (1) entering *Fast Recovery*, C is divided in half, and (2) a packet loss detected by the retransmission timer reduces C to zero. In the example of Figure 25, the unfriendliness will be easily detected during the transition from x_1 to x_2 and Reno's rules will be enforced, allowing the Vegas+ flow to obtain its fair share of network resources.

The Vegas+ solution does not try to solve the fundamental problems of Vegas discussed in Section II-G. Moreover, the reactive congestion control elements of Veagas+ practically nullify the inherited advantages of Vegas. Additionally, it has not been proven that Vegas+ will always stay in a moderate mode if there are no Reno flows present but the network is experiencing some kind of anomaly.

I. TCP Veno

Fu and Liew [18] propose a modification to the Reno congestion control algorithm (Section II-C) aimed at improving the throughput utilization of TCP. The key idea is to use the Vegas bottleneck buffer estimation technique to perform early detection of the congestion state. Unlike Vegas, this buffer estimation is used only to adjust the increase/decrease coefficient of the Reno congestion control algorithm, and thus does not inherit Vegas' problems.

The Veno (VEgas and reNO) algorithm defines two modifications. First, it limits the increase of the congestion window during the congestion avoidance phase if the Vegas buffer estimate shows excessive buffer utilization (i.e., $\Delta > \beta$). In other words, if the Vegas estimate indicates a congestion state, the sender starts probing network resources very conservatively (increasing by one for every two RTT, "*B*" in Figure 26). Second, reducing the congestion window upon entering *Fast Recovery* is modified to halve the $cwnd$ value only if the buffer estimate also indicates congestion. That is, in the event of detecting a loss and $\Delta > \beta$, the congestion window will be halved. Otherwise, if only a loss is detected, it will be reduced to 80% of its current size.

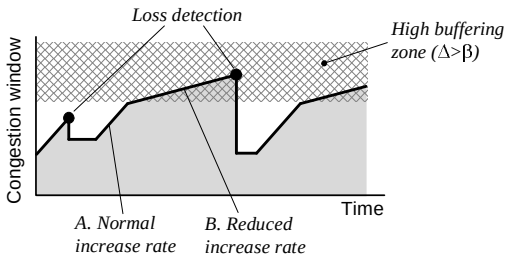


Fig. 26. Congestion window dynamics of TCP Veno

To summarize, the effectiveness of the Veno algorithm is slightly improved in comparison to Reno. Veno flow tends to stay longer in the congestion avoidance state with larger congestion window values. However, the price for this is additional latency to discover network resources. The Veno modification has practically no effect on fairness. Therefore we can consider it to have the same characteristics as the base Reno algorithm.

J. TCP Vegas A

Besides the inability to compete with Reno flows effectively (see Section II-H), TCP Vegas has a number of other internal problems [19]. For instance, under certain circumstances, Vegas can inappropriately choke off the flow rate to nearly zero. This happens because the assumption that the RTT will change only due to buffering is not entirely true. In fact, if the RTT increases due to a routing change, the algorithm will make a wrong decision, leading to the reduced flow rate. To illustrate, Figure 27 presents two curves, one for a low-RTT/low-rate (1, for example a DSL link) and another for a high-RTT/high-rate (2, for example a satellite link) path. If a route changes from 1 to 2 when the congestion window size is equal to $cwnd$, the algorithm will wrongly calculate buffering Δ_2 and may exceed a threshold. The minimal RTT for the low-RTT/low-rate link will be erroneously used as a baseline in calculating the expected rate for the high-RTT/high-rate link. That is, having a congestion-free state, the estimate indicates congestion.

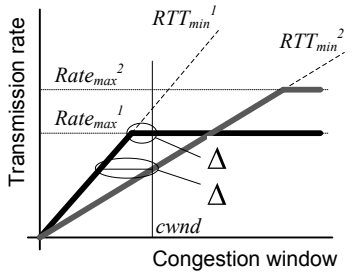


Fig. 27. TCP Vegas—estimation error if the path has been rerouted

Another assumption that surfaces occasionally and is incorrect is that all flows competing along the same path will observe the same RTT_{min}. Let us consider a situation with two Vegas flows, one that has been transmitting data for a long time and the other which has just started transmitting. Naturally, the long-lived flow has more chances to observe the true minimal RTT, compared to the new flow. The difference between the minimum RTTs that the two flows observe causes a difference in congestion state estimates (Figure 28): the old flow thinks that the network is congested while the new one estimated a congestion-free network state. As a consequence, the distribution of network resources favors the new flow.

Srithith et al. [19] have presented the VegasA (Vegas with Adaptation) algorithm, which extends the original Vegas congestion control with an adaptable mechanism. The threshold coefficients α and β from the Vegas algorithm are adjusted depending on the steady state dynamics of the actual transmission rate. That is, if VegasA detects an increase in the actual bandwidth while the system is in a stable state (i.e., $\alpha < \Delta < \beta$), it assumes a path change and shifts the boundaries of the control dead-zone upward ($\alpha = \alpha + 1$ and $\beta = \beta + 1$). The boundaries are shifted downward if some network anomaly is detected; for example, if the estimate is

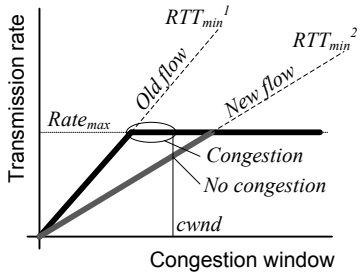


Fig. 28. TCP Vegas—estimation error if a new flow is observing a higher RTT_{min}

showing congestion-free state $\Delta < \alpha$, yet the rate in actuality has decreased. Additionally, boundaries are shifted downward every time the estimate shows the actual congestion.

Besides the threshold adaptability, VegasA adds additional conditions to the congestion window management algorithm. An increase is allowed in three cases: (1) if the estimate shows no congestion and a lower threshold α has a minimal value (the original Vegas rule); (2) if the actual rate has increased and the estimate is showing no congestion ($\Delta < \alpha$); or (3) the actual rate has decreased while the flow is in a steady state ($\alpha < \Delta < \beta$). A decrease should occur if either the network has been determined to be in a congestion state ($\Delta > \beta$) or if the actual flow rate has decreased and the network has been determined to be congestion-free.

According to simulation results [19], VegasA has substantial improvements in various aspects when compared to the original Vegas design. It preserves the Vegas properties of stabilizing throughput in a steady state and does not suffer significantly in the long term from changes in path RTT. To some extent, a VegasA flow can compete with Reno flows and acquire its resource share. However, these were only testing environments; the algorithm has not been evaluated in real networks. Moreover, new problems that are discussed in the next sections of this survey (such as the scalability issue in high-speed networks, resistance to random losses, etc.) give us reason to believe that VegasA is not a universal replacement for the Reno or NewReno congestion control algorithms.

III. Packet Reordering

All the congestion control algorithms discussed in the previous section share the same assumption that the network generally does not reorder packets. This assumption has allowed the algorithms to create a simple loss detection mechanism without any need to modify the existing TCP specification [1]. The standard already requires receivers to report the sequence number of the last in-order delivered data packet each time a packet is received, even if received out of order [21]. For example, in response to a data packet sequence 5,6,7,10,11,12, the receiver will ACK the packet sequence 5,6,7,7,7,7). In the idealized case, the absence of reordering guarantees that an out-of-order delivery occurs only if some packet has been lost. Thus, if the sender sees several ACKs carrying the same sequence numbers (duplicate ACKs), it can be sure that the network has failed to deliver some data and can act accordingly (e.g., retransmit lost packet, infer a congestion state, and reduce the sending rate).

Of course in reality, packets are reordered [26], [27]. This means that we cannot consider a single duplicate ACK (i.e., ACK for an already ACKed data packet) as a loss detection mechanism with high reliability. To solve this problem of a false loss detection, a solution employed as a rule of thumb establishes a threshold value for the minimal number of duplicate ACKs required to trigger a packet loss detection (e.g., three) [22], [11], [12]. However, there is a clear conflict with this approach. Loss detection will be unnecessarily delayed if the network does not reorder packets. At the same time, the sender will overreact (e.g., retransmit data or reduce transmission rate needlessly) if the network does in fact reorder packets.

Packet reordering can stem from various causes. For example, it can be erroneous software or hardware behavior, such as bugs, misconfigurations, or malfunctions. But packets can also be reordered in some networks as a side effect of a normal delivery process. For example, packets can be reordered if a router enforces diverse packets handling services (differentiated services [28], [29]) and internally reschedules packets in its queue (active queue management [30], [31]). Also if the network provides some level of delivery guarantees (e.g., wireless networks), the underlying layer (physical or link layer) can retransmit some portion of the data without TCP's prompting and cause a shuffling of the upper layer packets. Finally, channel bundling and packet processing parallelism will likely contribute a good portion of the future Internet [32], [33], [34].

In this section we present a number of proposed TCP modifications that try to eliminate or mitigate reordering effects on TCP flow performance (Table II). All of these solutions share the following ideas: (a) they allow nonzero probability of packet reordering, and (b) they can detect out-of-order events and respond with an increase in flow rate (optimistic reaction). Nonetheless, these proposals have fundamental differences due to a range of acceptable degrees of packet reordering, from moderate in *TD-FR* to extreme in *TCP PR*, and different baseline congestion control approaches. The development of these proposals is highlighted in Figure 29.

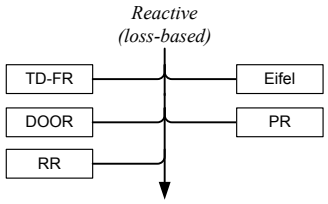


Fig. 29. Evolutionary graph of TCP variants that solve the packet reordering problem

A. TD-FR

A number of measurements conducted in the mid 1990s [26] proved the presence of out-of-order packet delivery in the Internet. This highlights the problem of potentially over-penalizing a TCP flow if its congestion control mechanism employs loss detection using duplicate ACKs. That is, in the

TABLE II
FEATURES OF TCP VARIANTS THAT SOLVE THE PACKET REORDERING PROBLEM

TCP Variant	Section	Year	Base	Added/Changed Modes or Features	Mod[1]	Status	Implementation		
							BSD[2]	Linux	Sim
TD-FR [26]	III-A	1997	Reno	Time delayed fast recovery	R	Experimental			
Eifel [35], [36]	III-B	2000	NewReno	Differentiation between transmitted and retransmitted data packets	S or (S+R+P)	Standard	i3.0, F*	2.2.10*	ns2
TCP DOOR [37]	III-C	2002	NewReno	Out-of-order detection and feedback, temporary congestion control disabling and instant recovery	S+R+P	Experimental			ns2*
TCP PR [38]	III-D	2003	NewReno	Fine-grained retransmission timeouts, no reaction to DUP ACKs	S	Experimental			ns2
DSACK [39]	III-E	2000	SACK	Reporting duplicate segments	R	Standard		>2.4.0	
RR-TCP [40], [41]	III-F	2002	DSACK	Duplicate ACK threshold adaptation	S	Experimental			ns2

[1] TCP specification modification: S = the sender reactions, R = the receiver reactions, P = the protocol specification
[2] i for BSDi, F for FreeBSD * optional or available in patch form

absence of the congestion or packet losses, each event of packet reordering triggers at least one (and probably more) duplicate ACKs, which can be considered an indication of congested pathways and a guide for reducing the transmission rate.

At the same time, there are two observations about out-of-order packet delivery. According to Paxson's work [26], this effect is not uniformly distributed across network sites. Measurements identified a low level of reorderings (0.1%–2% on average), with peaks in some traces as high as 36%. Moreover, Paxson made the most interesting observation: the data transfers having the highest degrees of reordering also experienced almost no packet losses.

Paxson [26] proposed a simple way to eliminate the penalties of reordering through TD-FR, time delayed *Fast Recovery*. If a receiver does not respond immediately to out-of-order data packets with duplicate ACKs, but postpones the action (e.g., by 8–20 msec, depending on the reordering pattern), a majority of the reordering events will be hidden from the sender. However, the advantage of this solution is, at the same time, a disadvantage as well. The artificial delay, aimed at preventing overreaction, adds to the time required to detect actual losses. If the delay grows too big, the "fast" loss-detection mechanism becomes slower than a conventional loss detection based on RTO. Clearly, the nondeterministic nature of the reordering effect demands some path adaptation mechanisms, which unfortunately are not implemented in Paxson's solution.

B. Eifel Algorithm

Ludwig and Katz [35] introduced the Eifel[2] algorithm as an alternative method to alleviate the negative effects of packet reordering in TCP throughput. Instead of the TD-FR approach of introducing additional delay to the loss detection process based on duplicate ACKs (Section III-A), Eifel tries to distinguish reordering and real loss events. It does not try to guess the event type upon reception of the first duplicate, but rather postpones the decision until the first non-duplicate ACK is received. In other words, if the TCP sender receives a number (e.g., 3) of duplicate ACKs, as in NewReno, it enters *Fast Recovery*. When a non-duplicate ACK is received, Eifel checks its content and makes a decision whether to continue Fast Recovery or abort recovery and restore the original congestion window value. The advantage of the Eifel algorithm is clearly visible in Figure 30. On the one hand, the defined actions of Eifel do not affect normal operations of the base congestion control algorithm when there is no packet reordering. On the other hand, when some packets are reordered, the original sending rate will be restored very quickly.

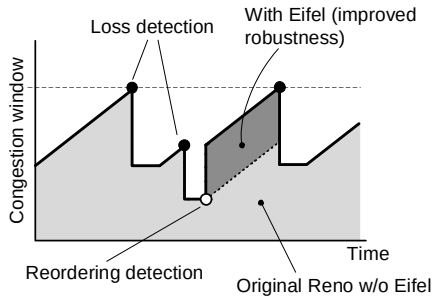


Fig. 30. Comparison of congestion window dynamics between Reno (NewReno) and Eifel

To reach the right decision, Eifel must resolve the ambiguity of a retransmission [20]. To clarify, let us consider a situation where a TCP sender decides to retransmit a data packet (e.g., due to receiving several duplicate ACKs). After receiving the first non-duplicate ACK, the sender does not know whether the retransmission helped resolve the problem or whether the problem resolved itself (as in case of a long burst of reordered packets). If either ACKs carry additional information to indicate not only a sequence number, but also some identification of the actual transmission, or if the ACKs can indicate that the ACK itself has been triggered by a retransmitted data packet, the ambiguity problem is easily resolved. The latter case is the easiest and most "cost-effective" way. For example, we can assign two bits from the unused space in the TCP header, where one bit is used to indicate retransmission of a

[2] Authors choice of spelling, after a mountain range in western Germany

data packet and the other one to echo this information back to the sender in an ACK. Although theoretically possible, a change in the TCP protocol is highly undesirable, as it makes deployment practically impossible.

We could instead use a standardized and highly deployed TCP Timestamp option [24]. In this case, the sender maintains an additional state variable (a time of the first retransmission) for each retransmitted data packet. Having the Timestamp option, each ACK packet will explicitly indicate what we need. If a received non-duplicate ACK has a timestamp less than a corresponding state variable, the sender can be sure that no actual losses have occurred on the path and transmission should be returned to the original state. This ability to protect TCP transfer from the packet reorderings in Internet paths, achieved with relative simplicity, allowed Eifel to become an RFC standard in 2005 [36].

C. TCP DOOR

Wang and Zhang [37] were concerned with TCP performance in mobile ad hoc networks (MANETs), which feature route changes with high probability and thus are highly penalized by the conventional congestion control algorithms. During route changes many packets can be lost, causing congestion control to make the wrong decision for reducing the rate of flow. If we can identify a time interval during which the network route has changed, then we can eliminate the penalty in TCP throughput by temporarily disabling the congestion control actions during this interval. This idea underlies the proposed TCP DOOR (Detection of Out-of-Order and Response).

During a route change event it is very probable that the order of IP packets will be changed. Thus, the problem to identify a route change can be replaced by identifying an out-of-order packet delivery. Similar to Eifel (Section III-B), in order to detect packet reordering reliably, each data and ACK packet should carry some additional information. For example, this information can be included in a new TCP option in the form of a special counter, which is increased every time a new packet is sent. In this case, the receiver can easily detect reordering and report it to the sender using some bit, either in the TCP header or in a new TCP option field. Another variant considered in paper [37] is utilizing a well-known Timestamp option [24] in a manner similar to the standard Eifel algorithm.

A reaction to detecting packet reordering (which can be considered an equivalent to route change in MANETs) entails two components (Figure 31). First, congestion control should be temporarily disabled to mitigate transition effects (time period T_1 in Figure 31). Second, if congestion control has recently reduced the sending rate due to loss detection (during time interval T_2 in Figure 31), the original state (the congestion window and retransmission timeout values) should revert (so called *instant recovery*). This action alleviates previous penalties from the detected rerouting event. The interval for the temporary congestion control disabling and the preceding time period for the instant recovery are not known a priori and depend on the actual network. Wang and Zhang conducted a number of simulations where they varied underlying routing protocols, timing values, and network conditions. Although some of the results show more than a 50% throughput improvement compared to TCP with the SACK option, there are cases with minimal to zero improvement.

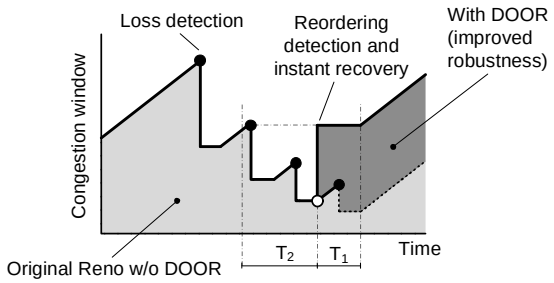


Fig. 31. Comparison of congestion window dynamics between Reno (or SACK) and DOOR

D. TCP PR

Bohacek et al. [38] noticed that since packet reordering is a common event in the network (e.g., in mobile ad hoc networks), duplicate ACKs cannot be considered reliable indications of either loss in the path or of congestion. In TCP PR (Persistent Reordering), the authors no longer assume the validity of inferring something from duplicate ACKs. Instead, they focus on making the retransmission timeout a robust and reliable loss and congestion indicator in a wide range of network environments.

In contrast to previously developed congestion controls, TCP PR maintains a timestamp for each transmitted data packet. A loss is detected whenever the timestamp of a data packet becomes older than the estimated RTT maximum (M). The concept of RTT maximum is similar to the RTO, but differs in implementation. Instead of the RTO recalculation once per RTT, the maximum estimate M is readjusted on each ACK arrival according to the formula:

$$M = \beta \cdot \max\left\{\alpha^{\frac{1}{cwnd}} \cdot M, RTT\right\}$$

where α and β are constants ($0 < \alpha < 1$, $\beta > 1$). Taking into account that a recalculation is made with each ACK, α represents a maximal decrease rate of the M in RTT timescale (i.e., $cwnd \times \alpha^{\frac{1}{cwnd}}$ is α).

As long as timeouts are treated optimistically (i.e., after a loss detection, flow is allowed to transmit at a multiplicatively reduced rate), TCP PR faces the problem of overreaction to multiple losses from the same congestion event, similar to Reno. To resolve this, each transmitted packet is also tagged with the current value of the congestion window. When a packet is lost, the congestion window is reduced by no more than half of the stored value for the lost packet.

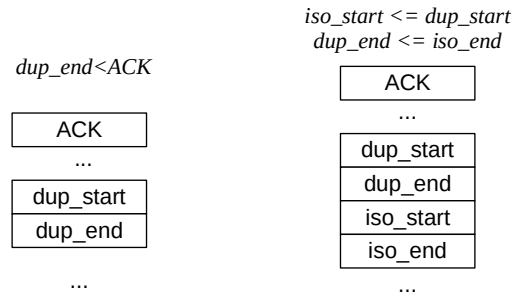
Because of different loss detection mechanisms, we cannot directly compare TCP PR with the previous congestion control algorithms. If we assume that an algorithm based on fine-grained timeouts is as robust as one based on duplicate ACKs, the fairness and effectiveness characteristics will be exactly the same as presented in Section II-C. Though this is not entirely true in all network environments, there are networks (e.g., MANETs), where duplicate ACKs are highly

unreliable feedback. Thus, TCP PR can greatly help improve TCP efficacy in those cases, e.g., where the network normally reorders packets.

E. DSACK

The specification of the selective ACK extension for TCP [14] does not define particular actions to take if a receiver encounters a data packet which has already been delivered. This can happen, for example, if the network reorders or replicates data packets, or if the sender wrongly estimates the retransmission timeout. The DSACK (Duplicate Selective ACKnowledgements) specification [39] complements the standard and provides a backward-compatible way to report such duplicates.

DSACK requires the receiver to report each receipt of a duplicate packet to the sender. However, there are two possibilities of duplication, which should be treated in slightly different ways. First, the duplicated data can be some part of the acknowledged continuous data stream. Second, it can be a part of some isolated block. In the former case, a DSACK-compliant receiver should include a range of sequence numbers in the first block of the SACK option (Figure 32a). In the latter case, besides including a duplicate range in the first block, the receiver should attach the isolated block at the second position in the SACK option (Figure 32b). In that way, DSACK, without violating the SACK standard [14], provides a way to report packet duplication.



(a) DSACK report (b) DSACK+SACK report

Fig. 32. DSACK reporting

Similar to SACK, the DSACK specification does not specify any particular actions for the sender. Instead, its authors merely discuss several issues for future research. One such issue is the detection of packet reordering events. If a sender can assume that duplication is caused primarily by packet reordering, it can undo some previous congestion control actions upon receipt of a DSACK packet (similar to Eifel and DOOR). Other discussed issues include a differential treatment of the normal SACK and DSACK packets, an implementation of some form of an ACK congestion control, and resolving the issue of the RTO underestimation. However, DSACK-based solutions should not blindly trust the DSACK information, as the receiver can send faulty information, either intentionally or unintentionally.

F. RR TCP

The SACK option by itself can provide a lot of information about patterns of packet delivery. For example, the occurrence of a reordering event can be detected if the sender receives a selective ACK packet followed by a cumulative ACK. Moreover, in this case we can also calculate a reordering length, i.e., how long a packet was delayed, in terms of packets. However, this would work only if no packets were retransmitted. Otherwise, it is unknown which event (original or repeated transmission) might have helped to recover a packet loss previously reported by the SACK. An approach presented in RR TCP (Reordering Robust) [40], [41] uses a DSACK to resolve the retransmission ambiguity. In short, after a sender retransmits a data packet that has been detected as lost, a succession of an ACK (or a SACK) and a DSACK, both covering the retransmitted packet, indicates that both the original transmission and the retransmission were actually successful. Because the sender knows the exact transmission sequence (the order packets were transmitted and retransmitted), reordering length can be easily calculated. The downside of this approach is that we cannot infer anything if either of the first or second ACK is lost.

RR TCP defines a way to use the calculated reordering length. If we know how long packets are usually delayed, we can adjust a threshold of duplicate ACKs (*dupthresh*, which usually is 3), which triggers the Fast Recovery phase. This, in contrast to Eifel (Section III-B) or DOOR (Section III-C), will proactively protect the sender from overreacting if packets have been reordered, not lost. Unfortunately, if *dupthresh* is set too high, all advantages of the robust loss detection will be eliminated. RR TCP includes a concept of a controlling loop for finding the optimal *dupthresh* value for a given path using a combined cost function, which integrates several costs including false timeouts and fast retransmits. Experimental evaluation shows consistent improvements with RR TCP, compared to TCP with the SACK option, in a wide range of network environments (i.e., varying delays, loss ration, reordering lengths). However, these improvements are effective only in long-lived TCP connections.

IV. Differential Services

Different application types have different data transfer requirements. Some applications, composing one group, have strict requirements for request-response delay and throughput (e.g., WEB browsing, FTP transfer, etc). Other applications do not have any particular requirements and are highly tolerant of the network conditions (e.g., automatic updates). In general, if traffic of the first application group can be prioritized, the overall user-perceived quality of service in the network (QoS) can be increased [42]. Unfortunately, due to a high level of Internet heterogeneity, even though there have been a number of attempts to provide a QoS functionality on the network (IP) level [43], [44], this feature is not yet globally available [45]. To overcome the deployment problem and yet provide some level of QoS, two host-to-host TCP-based prioritization techniques have been proposed (Table III).

The central idea among the proposed solutions is to enforce and guarantee, through special congestion control policies,

an "unfair" network share distribution between high- and low-priority flows. This idea may seem to contradict the basic fairness requirement for TCP congestion control: a new congestion control should not be more aggressive than the standard TCP congestion control algorithms (Reno, NewReno, and SACK). However, if we restrict the TCP-based QoS scope only to a low-priority service (i.e., to a problem of finding congestion control policies that would guarantee the network resource release if there are high-priority—standard TCP—flows present), then we definitely will comply with the fairness requirement.

In the remaining part of this section we will provide an overview of the two existing TCP-based QoS proposals (Table III), which share the idea of providing a one-level, low-priority data transfer service. The key differences between proposals are: (a) different baseline congestion control algorithms (Vegas for Nice and Reno for LP, see Figure 33), and (b) different mechanisms to detect the presence of a high-priority data transfer.

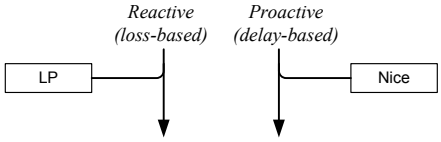


Fig. 33. Evolutionary graph of TCP variants that implement a low-priority data transfer service

A. TCP Nice

Venkataramani et al. [46] identified the need to optimize the network resources in the presence of a large number of background transfers—automatic updates, data backups, peer-to-peer file sharing, etc. As a solution, they proposed a new congestion control algorithm, TCP Nice, that enables a simple distributed host-to-host mechanism to minimize the interference between high-priority (foreground) and low-priority (background) flows. More particularly, Nice's congestion control policies are adjusted to react highly conservatively to all detected network state changes. In one sense, Nice considers all standard TCP flows as carrying high-priority data and tries to consume the network resources only if nobody else uses them.

The design of Nice is based on the Vegas algorithm (see Section II-G). There are two main reasons for this choice: (1) Vegas incorporates a proactive congestion detection mechanism which allows redistributing network resources between competing TCP flows without inducing any packet losses (i.e., interference between Vegas flows is lower than that for standard TCP flows); (2) due to its proactive nature, a TCP flow running the Vegas congestion control algorithm has problems capturing its network resource share while competing with a reactive Reno-like TCP flow (i.e., standard TCP flow), i.e., Vegas itself provides some level of a low-priority data transfer service.

To provide a guarantee of the transmission rate reduction in the presence of standard TCP flows, Nice defines a concept similar to the *queueing delay* threshold defined in the DUAL algorithm (Section II-B). However, there are several major differences. First, the queuing delay is compared to the threshold upon the arrival of each non-duplicate ACK packet. Second, instead of the averaged RTT, a current RTT sample is used in queuing delay calculations. Finally, the occurrence of a current queuing delay estimate exceeding the threshold does not automatically trigger changes in the congestion windows. Instead, Nice counts the number of times (X) that the queuing delay exceeds the threshold during each RTT period:

$$\forall t \in (t_0, t_0 + RTT) \quad Q_{ACK}(t) > Q_{thresh} \;\Rightarrow\; X = X + 1$$

The counted value X estimates the number of ACK packets which have been delayed due to interference with cross traffic (e.g., high-priority flows). If we assume the idealized case when no ACKs are lost or delayed by the receiver, then the ratio between X and the congestion window (measured in packets) would estimate a percent of enqueued (delayed) packets during the latest RTT. In Nice, if this estimate exceeds a predefined threshold, the congestion window is halved (Figure 34). The right choice of threshold value can make Nice much more sensible than both the original DUAL and Vegas algorithms. In addition, Nice allows the congestion window size to be a fraction (the minimum is $1/48$), meaning that only one packet is allowed to be sent in several RTT periods (48 RTTs in the worst case). This makes Nice even more conservative in network resource utilization in the presence of cross traffic.

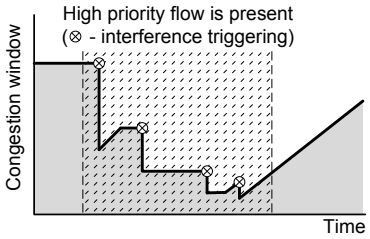


Fig. 34. Congestion window dynamics of TCP Nice

B. TCP LP

Almost concurrently with the Nice algorithm proposal (see Section IV-A), Kuzmanovic and Knightly [47], [42] presented a similar algorithm, TCP LP (Low Priority). It aimed to provide a low-priority data transfer service for background applications (e.g., software updates, data backup, etc.). However, for the baseline congestion control algorithm, its authors have chosen NewReno instead of Vegas. Other differences are in the way the presence of cross traffic is detected and what preventive measures are applied to minimize interference.

In TCP LP the DUAL's calculation of a queuing delay (see Section II-B) is refined progressively by using more accurate delay estimates. For this purpose LP makes use of the Timestamp option [24] and applies heuristics to estimate the one-way propagation delay (e.g., similar to Choi and Yoo's proposal [48]). Although this can complicate queueing delay calculations, the resulting values are much more resistant to congestion in the reverse channel, thus the level of false

TABLE III
FEATURES OF TCP VARIANTS THAT IMPLEMENT A LOW-PRIORITY DATA TRANSFER SERVICE

TCP Variant	Section	Year	Base	Added/Changed Modes or Features	Mod[1]	Status	Implementation	
							Linux	Sim
Nice [46]	IV-A	2002	Vegas	Delay threshold as a secondary congestion indicator	S	Experimental	2.3.15*	
LP [47], [42]	IV-B	2002	NewReno	Early congestion detection	S	Experimental	>2.6.18	ns2

[1] TCP specification modification: S = the sender reactions, R = the receiver reactions, P = the protocol specification
* optional or available in patch form

congestion detections is substantially decreased. The actual process of congestion detection (in terms of LP it is *early congestion detection*) with minor modifications repeats the one defined in DUAL: (1) LP maintains minimum and maximum one-way delays during the connection lifetime, and (2) once every RTT, TCP LP compares the current one-way delay estimate with a predefined threshold (a fraction of queuing delay plus a minimum of one-way delay).

The unique feature of the TCP LP algorithm is its reaction to *early congestion detection*. Upon detection of a first such event, LP reduces the congestion window to half the current value and starts the *inference timer*. If the sender triggers another early congestion detection event before the timer elapses, LP infers presence of the high-priority flow and the congestion window is reduced to the minimal value. In other cases, LP resumes the normal (Reno-like) congestion avoidance actions (Figure 35).

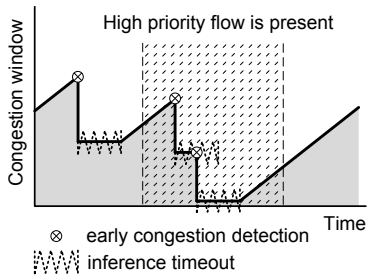


Fig. 35. Congestion window dynamics of TCP LP

NS2 simulations and real-world experiments using a Linux implementation of the LP algorithm have shown that it indeed has the desired property of yielding network resources to the standard TCP (high-priority) flows and, at the same time, successfully utilizing the network bandwidth if no such flows are present. Moreover, LP is able to fairly distribute the network resources among low-priority flows (inter-fairness).

There is no definitive answer to whether the TCP LP algorithm or Nice algorithm is better. On one hand, both of them are extremely sensitive to activity in the network, and thus fulfill a necessary condition for the low-priority service implementation. But on the other hand, there is a big question as to how well both algorithms are able to utilize the network capacity if only low-priority flows are present. Although Nice and LP should have the same characteristics as the baseline algorithms (Vegas and NewReno respectively), this has not been proved. Moreover, widespread use of wireless and high-speed networks limit the applicability of either Nice or LP, due to ineffectiveness of the baseline algorithms in those environments. Although Kuzmanovic et al. [49] made an attempt to create a high-speed modification of LP, HSTCP-LP, additional research is required to investigate a real-world applicability of the designed solution.

V. WIRELESS NETWORKS

The growing spread of wireless networks has highlighted the need for TCP protocol modification. Originally designed for wired networks where congestion is the primary cause of packet losses, TCP is unable to react adequately to packet losses not related to congestion. Indeed, if a data packet is lost due to short-term radio frequency interference, then there are no router buffer overflows and TCP's decision to reduce the congestion window is wrong. Instead, it should just recover from the loss and continue the transmission as if nothing had happened.

Several solutions have been proposed to resolve this problem. One group gives up the idea of a pure host-to-host data transfer either by (a) requiring routers to disclose the network state (e.g., using explicit congestion notification [50]), by (b) relying on network channels to recover from the non-congestion-related losses (e.g., link-layer retransmission [51] or TCP packet snooping and loss recovery by intermediate routers [52]), or by (c) isolating the wireless error-prone and wired error-safe transmission paths using an intermediate host [53], [54]. These approaches are beyond the scope of this survey and have been thoroughly discussed by Lochert et al. [2].

In this section we focus on solutions that keep the host-to-host idea and at the same time provide some level of resistance to non-congestion related packet losses. The bandwidth estimation technique proposed by Mascolo et al. as a part of TCP Westwood [55] laid the foundation for the sender-side distinguishing between a congestion-related and an unrelated (random) loss without any support from the network. Research that follows (Figure 36) identified several of Westwood's weaknesses, for example, bandwidth overestimation, insufficient robustness in networks with extreme levels of transmission errors, etc. Table IV shows characteristic features of refinements in Westwood that try to mitigate discovered problems.

A. TCP Westwood/Westwood+

TCP Westwood proposed by Mascolo et al. [55] keeps the distributed network-independent ideology of TCP and is a

TABLE IV
FEATURES OF TCP VARIANTS THAT ENABLE RESISTANCE TO RANDOM LOSSES

TCP Variant	Section	Year	Base	Added/Changed Modes or Features	Mod[1]	Status	Implementation	
							Linux	Sim
TCP Westwood [55], [56]	V-A	2001	NewReno	Estimate of available bandwidth (ACK granularity), Faster Recovery	S	Experimental		ns2*
TCP Westwood+ [57]	V-A	2004	Westwood	Estimate of available bandwidth (RTT granularity)	S	Experimental	>2.6.12	
TCPW CRB [58]	V-B	2002	Westwood	Available bandwidth estimate (combination of ACK and long-term granularity), identifying predominant cause of packet loss	S	Experimental		ns2*
TCPW ABSE [59]	V-C	2002	CRB	Available bandwidth estimate (continuously varied sampling interval), varied exponential smoothing coefficient	S	Experimental		ns2*
TCPW BR [60]	V-D	2003	Westwood	Loss type estimation technique (queuing delay estimation threshold, rate gap threshold), retransmission of all outstanding data packets, limiting retransmission timer backoff	S	Experimental		
TCPW BBE [61]	V-E	2003	Westwood	Effective bottleneck buffer capacity estimation, reduction coefficient adaptation, congestion window boosting	S	Experimental		

[1] TCP specification modification: S = the sender reactions, R = the receiver reactions, P = the protocol specification
* optional or available in patch form

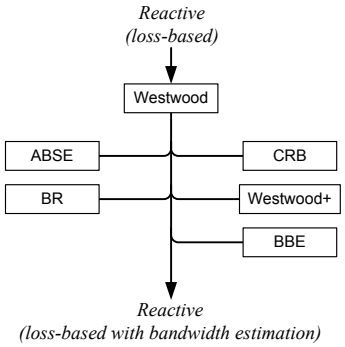


Fig. 36. Evolutionary graph of TCP variants that enable resistance to random losses

modification of the NewReno TCP congestion control algorithm. At the same time, it can significantly improve the data transfer efficiency in error-prone networks (e.g., wireless). To do so Westwood replaces the blind Reno's congestion control actions that are triggered by loss detection (i.e., halving if three duplicate ACKs are received) with a heuristic-based procedure of setting the congestion window w to an optimal value (*Faster Recovery*). As an optimum, the heuristic considers a value which corresponds to a data transfer rate observed in the recent past ($w \approx rate \times RTT$). Indeed if there is a random error due to wireless interference, the optimum would reflect the best choice for the sender: transmission without any rate reduction. In another case, if a packet is lost due to congestion in the network, the data reception rate recently observed by the receiver is exactly the rate at which the network is capable of delivering data from the sender ("achieved data rate"). If the sender continues transmission at a rate equal to that observed by the receiver, the number of newly transmitted packets will be equal to the number of delivered packets (router queues would not be growing), and additional congestion will be prevented.

Having this win-win situation for all packet loss cases, the only question is how the sender can discover the rate observed by the receiver. As a direct solution we can ask the receiver to send special rate notifications. However, from the deployment point of view, this is extremely hard. The proposed [55] and later patented [56] solution is to perform a sender-side estimation of the actual delivery rate based on an existing notification mechanism (i.e., using ACK packets).

To illustrate the rationale behind this estimate, let us consider the following example (Figure 37). If we assume that an ACK packet is generated right after a data packet is received and that ACKs are evenly delayed in the return path, the ACK rate observed by the sender will be equal to the data delivery rate observed by the receiver. To calculate the forward-path bandwidth actually utilized, we just need to multiply the ACK rate by the amount of acknowledged data. The bandwidth calculation holds in the long term even if some ACKs are lost or delayed by the receiver; i.e., a decrease in the ACK rate will be compensated by an increase in the acknowledged data amount.

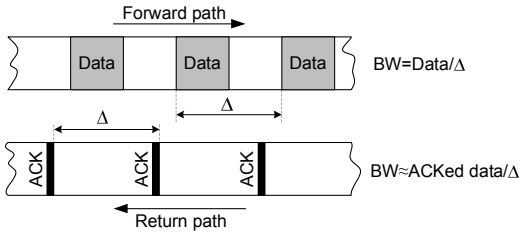


Fig. 37. Rationale for the available bandwidth estimation technique

To mitigate fluctuations, Westwood has a two-level bandwidth estimate processing capability. On the first level, the instantaneous estimate is calculated upon reception of an ACK packet ($b = d/\Delta$, where d is the amount of acknowledged data by the ACK and Δ is the time elapsed since the last ACK received). On the second level, the calculated instantaneous

values are averaged with a special discrete time filter [55]:

$$B = \alpha(\Delta) \cdot B^{-1} + [1 - \alpha(\Delta)] \cdot \left(\frac{b + b^{-1}}{2} \right)$$

where $\alpha(\Delta)$ is the averaging coefficient, as a function of Δ; b and b^{-1} are current and previous samples of the bandwidth estimate; and B^{-1} is the previously calculated average value of the estimate.

Although set-up experiments have shown a good level of precision for Westwood's estimate, practice has discovered that the calculation may be substantially wrong in certain network conditions [59], [57]. For example, in the presence of the ACK compression effect [62], when ACKs are differently delayed and grouped due to congestion over the reverse path, discrete averaging of instantaneous bandwidth estimation samples leads to substantial overestimation. For that reason, in the revised Westwood+ algorithm [57] the estimate has been changed so that it is calculated with RTT granularity; i.e., in the formula $b = d/\Delta$, d is now the amount of acknowledged data during the last RTT and Δ is the RTT itself. This estimate of average bandwidth during the last RTT is defined to be further averaged in long-term using the well-known exponential smoothing technique, with a smoothing factor $\alpha = 0.9$:

$$B = \alpha \cdot B^{-1} + (1 - \alpha) \cdot b$$

Although it has been asserted the Westwood algorithm shows good fairness properties, this is not entirely straightforward from a theoretical point of view. Presence of intra-fairness property (i.e., fairness between TCP flows running the Westwood algorithm) can be shown using the diagram in Figure 38. After two flows start competing from any state x_0, they increase their congestion window (i.e., share of network resources) evenly, until a network limit is reached. It can be shown that if two Westwood flows simultaneously detect a congestion event and reduce their congestion windows w based on the *achieved rate estimate* ($w = B \times RTT$), the ratio between flows' congestion windows would remain intact. During the consecutive *Congestion Avoidance* phase, the ratio would slowly increase (e.g., if upon loss detection the congestion window sizes of the two flows have the ratio 1:10, then after ten steps of linear increase—in ten RTTs—a new ratio would be 11:20). Clearly, in a finite number of steps, the ratio between congestion window sizes will become very close to one.

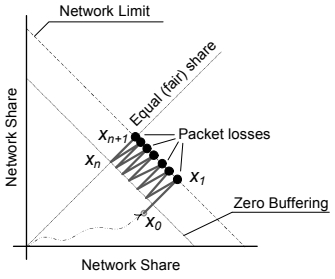


Fig. 38. Convergence diagram when two Westwood flows are competing with each other

Unfortunately, inter-fairness or fairness between Westwood and legacy Reno-type flows is not quite definite. In an idealized case, when a Westwood flow knows the exact amount of utilized network resources, a Reno flow will be suppressed. This will happen, in theory, because Reno always halves its network share while Westwood sets it depending on the estimate value. However, in practice, due to various random processes in the network and an imprecise bandwidth estimation technique (ACKs may be delayed or lost), Westwood/Westwood+ flows can compete successfully and relatively fairly with Reno-type flows.

B. TCPW CRB

Wang et al. [58] acknowledged the critical vulnerability of Westwood: under certain network conditions the bandwidth estimation (BE) technique gives highly inaccurate results. As a solution to this problem they proposed TCPW CRB (Westwood with Combined Rate and Bandwidth estimation), which refines the estimation algorithm by complementing it with a conservative long-term bandwidth calculation ("rate estimation" RE) technique. It is similar to one from the Westwood+ proposal, but the sampling period is some predefined constant T, instead of a measured RTT.

Experimental results show that the long-term estimate prevents overestimation if a network is experiencing congestion. At the same time, it is likely to underestimate bandwidth in the presence of random errors. To tackle both underestimation and overestimation problems simultaneously, CRB maintains two estimates, an old and a new. Upon detecting a packet loss, CRB chooses one of the estimates depending on the assumed predominant loss type: the old estimate for random loss (i.e., new value of the congestion window is calculated as $BE \times RTT_{min}$) and the new one for congestion loss (i.e., the congestion window is set to $RE \times RTT_{min}$).

A primary cause for loss is assumed to be congestion when the long-term bandwidth estimate RE shows a high level of imprecision, which is determined by comparing the ratio between the current congestion window size and relation $RE \times RTT_{min}$ to a predefined threshold θ. If this ratio is lower than the threshold θ (e.g., $\theta = 1.4$), a congestion event is assumed; otherwise, CRB thinks that loss is not related to congestion.

As long as CRB does not conceptually change Westwood policies upon detecting a loss (i.e., *Faster Recovery*), intra-fairness characteristics remain unchanged. CRB authors claim that the dual bandwidth estimate (BE and RE) improves Westwood fairness to legacy Reno/NewReno flows. However, this has been confirmed only through a number of NS2 simulations and the authors agree that future investigation is required to evaluate CRB in wide-range network scenarios that include real Internet experiments.

C. TCPW ABSE

As an extension of CRB (Section V-B), Wang et al. [59] proposed TCPW ABSE (Westwood with Adaptive Bandwidth Share Estimation). ABSE leverages the idea of dual bandwidth estimation by introducing a bandwidth sampling

interval adaptation mechanism. In other words, instead of two predetermined sample intervals for CRB (ACK inter-arrival and a long predefined constant period), ABSE continuously changes the interval depending on an estimated network state. The network state estimation heuristic is adapted to directly control the length of a sampling interval Δ in slightly changed form compared to CRB:

$$\Delta = \max\left(\Delta_{min}, \frac{RTT \cdot (VE - RE)}{VE}\right)$$

where Δ_{min} is a predefined minimal sampling interval, VE is a Vegas-type estimation of expected rate ($VE = cwnd/RTT_{min}$, see Section II-G), and RE is an exponentially averaged bandwidth estimate with a sampling interval equal to the RTT, similar to Westwood+ (Section V-A). If the current value of RE is significantly smaller than predicted by the Vegas-like estimation VE, the network is likely to be in severe congestion. Thus, similar to CRB, a long sampling interval will be calculated (i.e., Δ is close to RTT when $RE \to 0$). In the opposite case, when VE and RE are close (i.e., when a number of lost packets are close to or equal zero and when RTT is close to the minimal value), the minimal sampling interval will be used. Clearly, these observations of the border cases comply with the definition of the CRB heuristic. It is claimed that smooth adaptation of the sampling interval improves estimation precision in transition periods.

In addition to the adaptive calculation of a sampling interval, ABSE also defines a varied exponential smoothing coefficient for averaging bandwidth estimation samples. The basic idea is to make the averaging sharper if availability of network resources is changing very dynamically (the new sample should have a bigger impact on the averaged value), and smoother otherwise. The level of dynamics is calculated through a bandwidth estimate jitter. Through NS2 simulations, ABSE's authors have confirmed that the varied smoothing coefficient is able to help achieve a fast response to changes, and at the same time provide resistance to the noise.

Similar to CRB, ABSE does not change Westwood's concept of *Faster Recovery*, and thus has similar inter-fairness properties. In addition, NS2 simulations showed very good characteristics of ABSE fairness to legacy NewReno flows. However, real-world experiments are required to confirm simulation results.

D. TCPW BR

Though the Westwood approach (Sections V-A through V-C) can significantly improve the effective TCP throughput in the presence of non-congestion related packet losses, Yang et al. [60] discovered that it cannot effectively handle volumes of random errors ($> 2\%$). A newly proposed TCPW BR (Westwood with Bulk Repeat) algorithm is also based on Westwood, but additionally integrates a special loss-type detection mechanism. Upon each loss detection, if it is estimated to be non-congestion related, BR applies very aggressive (highly optimistic) recovery policies instead of the original ones. The proposed loss type estimation mechanism in BR is a compound of two loss-detection algorithms [63]: *the queuing delay estimation threshold* and *rate gap threshold* algorithms.

The queuing delay estimation threshold (QDET) algorithm is similar to *Spike* [64] and is based on the DUAL concept of measuring the queuing delay (Section II-B). The main difference is that QDET maintains two thresholds T_{start} and T_{end}. T_{start} represents a condition for entering the state when all losses are assumed to be caused by congestion ($T_{start} = \alpha \cdot Q_{max}$). T_{end} is a condition for returning to the default state when non-congestion loss type is assumed ($T_{start} = \beta \cdot Q_{max}$). The threshold coefficients α and β can be, for example, 0.4 and 0.05 respectively.

The rate gap threshold algorithm is based on comparing a Westwood's bandwidth estimate (BE) to a fraction α of the expected throughput (VE). The latter is calculated in a manner similar to Vegas (Section II-G): $VE = cwnd/RTT_{min}$. If Westwood's estimate is less than the predefined fraction of expected throughput, loss is assumed to be due to a congestion event; otherwise, non-congestion related loss type is assumed. The rationale behind this comparison is that when there is no congestion, even in the face of a substantial number of packet losses, the data throughput is still relatively close to the expected (i.e., RTT is close to RTT_{min} and the amount of delivered data packets during last the RTT is close to $cwnd$).

Yand et al. [60] claimed that utilizing two independent loss-type estimation mechanisms increases estimation precision and reduces the number of false positives. This is especially crucial because of BR's policies when detecting a non-congestion loss. If this is the case, BR immediately retransmits all data packets that have been transmitted and have not yet been acknowledged (outstanding data packets), and does not modify the congestion window size. In some environments these policies can be extremely helpful in the case of real non-congestion losses, and much more effective than TCP SACK/FACK (Section II-E, II-F) policies due to their internal limitations (i.e., one SACK can indicate no more than four blocks of lost packets).

In addition, BR changes the *retransmission timer backoff algorithm* (see Section II-A) by limiting a maximum timer value during non-congestion packet losses with a predefined constant. This decision improves the recovery time in environments where the probability of loss is extremely high.

E. TCPW BBE

Shimonishi et al. [61] showed that flows running the Westwood, Westwood+ (Section V-A), or ABSE (Section V-C) congestion control algorithms can be highly unfair to standard TCP flows if the network has limited buffering capabilities. To resolve this problem, they introduced BBE, a Bottleneck and Buffer Estimation algorithm that refines the Westwood policy of reducing the congestion window size w upon detecting a loss. More specifically, BBE complements the congestion window reduction policy with an additional variable coefficient $\mu \leq 1$: $w = RTT_{min} \times B \times \mu$, where B is the Westwood estimate of a recent achievable rate. This coefficient borrows DUAL's concept (Section II-B) of sensing the current network state: when the current queuing delay Q is close to a maximum Q_{max}, the network is considered to be experiencing congestion and μ should be $1/2$ (i.e., same as the standard

TCP during congestion); otherwise, the network is congestion-free, and μ can be 1. The actual proposed calculation of the coefficient μ is defined as $\mu = Q_{max}/(Q + Q_{max})$ where Q_{max} is not just the maximum queuing delay observed during the connection lifetime, but the exponentially smoothed queuing delay samples obtained just before each loss detection event. This technique allows BBE to adapt easily to network changes.

Additionally, BBE recognizes the overestimation problem in the original Westwood algorithm and proposes a hybrid estimation technique. In particular, BBE calculates the achievable rate as a weighted sum of two estimates: the variable sampling rate B_v (e.g., ACK rate, as in original Westwood) and a constant sampling rate B_c (e.g., 1/RTT as in Westwood+). The weighting is as follows: $B = \gamma \cdot B_v + (1 - \gamma) \cdot B_c$. The coefficient γ, similar to the above-mentioned additional reduction coefficient μ, relies on the queuing delay concept: $\gamma = 1/e^{\alpha \times RTT/RTT_{max}}$ where α is some large positive constant. As we can see, the estimate calculation follows a general observation: the constant rate sampling is more precise when the network is experiencing congestion (i.e., B_c has more weight when RTT is close to RTT_{max}), and the variable rate is more precise when the network is congestion-free (i.e., B_v has more weight when RTT is far from RTT_{max}).

The simulation results provided by BBE's authors have shown quite a good fairness to the legacy NewReno flows, along with a good data transfer performance comparable to the original Westwood algorithm. However, without further theoretical and practical investigations it cannot be claimed that BBE provides a universal solution for congestion control in wired-wireless networks. Moreover, appearance of the high-speed networks (both wired and wireless) has opened a number of other problems (see Section VI) which outweigh the issue of fairness to NewReno, under a wide variety of network conditions.

VI. High-Speed/Long-Delay Networks

The emergence of high-speed networks uncovered the inability of deployed TCP variants (Reno, NewReno, SACK, etc.) to use the resources of these networks effectively. All of the congestion control algorithms discussed in Sections II through V improve different aspects of efficiency for data transfer, without questioning the basic principle on which it rests, which was defined as far back as 1988 as part of Tahoe (Section II-A): network resource discovery during the congestion avoidance phase should be highly conservative. In TCP implementations, this principle was generally realized with a congestion window ($cwnd$) increase by one packet for each RTT if no errors were detected. This works quite well if network capacity or round-trip delays are relatively small, but does not work well otherwise.

To illustrate the problem—sometimes referred to as the *bandwidth-delay product problem* (*BDP problem*)—let us consider a TCP flow trying to discover all the resources of some network channel. The minimum time required for this, assuming there are no packet losses, is on the order of the channel *bandwidth delay product* (BDP). More precisely, to get to a theoretical upper bound of a TCP data transfer rate ($D \times cwnd/RTT$, where D is a maximum data packet), Reno/NewReno flow needs about $D \times cwnd$ RTTs, because $cwnd$ increases by one every RTT. In a network having 10 GiBit/s capacity, 100 ms round-trip delay, and a maximum data packet size of 1500 bytes, it would take almost two hours [65], [66]. Moreover, all the packets must be delivered during these two hours, and that is equivalent to an unrealistic packet loss probability.

In the remaining part of this section we discuss various solutions (Table V) that address several congestion control problems. Although these solutions rest on different assumptions and approaches (see Figure 39), they have the same objective: to create an ideal algorithm for high-speed (e.g., optical) or large delay (e.g., satellite) links. The algorithm should simultaneously (a) provide for efficient use of network resources, (b) respond quickly to network changes, and (c) be fair to other flows present in the network. The latter is divided generally into the three categories: (1) *intra-fairness*—characteristic of resource distribution between flows running the same congestion control algorithm in the same network environment; (2) *inter-fairness*—characteristic of distribution between flows running different algorithms in the same environments; and (3) *RTT-fairness*—characteristic of resource distribution between flows sharing the same bottleneck link but having different RTTs.

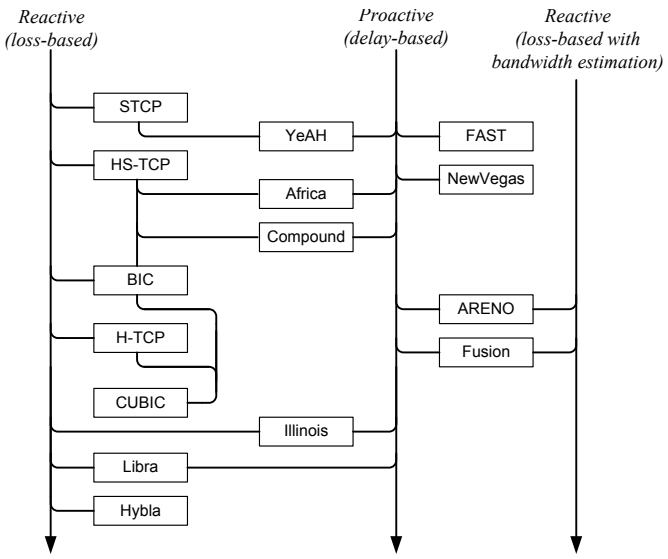


Fig. 39. Evolutionary graph of TCP variants aimed at improving efficiency in high-speed or long-delay networks

A. HS TCP

After recognizing TCP's efficiency problem in high-speed networks, Floyd [67], [65] proposed the HS-TCP (High-Speed TCP) algorithm. This is an experimental congestion control method that has several objectives. Among them are (a) efficiency in high bandwidth-delay product (BDP) networks, without relying on unrealistically low loss rates; and (b) fairness to standard TCP in high loss rate environments. For this purpose HS-TCP replaces the standard NewReno

TABLE V
FEATURES OF TCP VARIANTS AIMED AT IMPROVING EFFICIENCY IN HIGH-SPEED OR LONG-DELAY NETWORKS

TCP Variant	Section	Year	Base	Added/Changed Modes or Features	Mod[1]	Status	Implementation		
							Win[2]	Linux	Sim[3]
HS-TCP [67], [65]	VI-A	2003	NewReno	Additive increase steps and multiplicative decrease factors as functions of the congestion window size, Limited Slow-Start	S	Experimental		>2.6.13	ns2
STCP [66]	VI-B	2003	NewReno	Multiplicative Increase Multiplicative Decrease congestion avoidance policy	S	Experimental		>2.6.13	
H-TCP [68], [69]	VI-C	2004	NewReno	Congestion window increase steps as a function of time elapsed since the last packet loss detection, scaling increase step to a reference RTT, multiplicative decrease coefficient adaptation	S	Experimental		>2.6.13	
TCP Hybla [70]	VI-D	2004	NewReno	Scaling the increase steps in Slow-Start and Congestion Avoidance to the reference RTT, data packet pacing, initial slow-start threshold estimation	S	Experimental		>2.6.13	
BIC TCP [71]	VI-E	2004	HS-TCP	Binary congestion window search, Limited Slow-Start	S	Experimental		>2.6.12	ns2.6*
TCP Cubic [72]	VI-F	2008	BIC	The congestion window control as a cubic function of time elapsed since a last congestion event	S	Experimental		>2.6.16	ns2.6*
FAST TCP [73], [74], [75]	VI-G	2003	Vegas	Constant-rate congestion window equation-based update	S	Experimental			ns2.29*
TCP Libra [76]	VI-H	2005	NewReno	Adaptation of the packet pairs to estimate the bottleneck link capacity, scale the congestion window increase step by the bottleneck link capacity and queuing delay	S	Experimental			ns2
TCP NewVegas [25]	VI-I	2005	Vegas	Rapid window convergence, packet pacing, packet pairing	S	Experimental			
TCP AR [77]	VI-J	2005	Westwood, Vegas	Congestion window increase steps as a function of the achievable rate and queuing delay estimates	S	Experimental			
TCP Fusion [78]	VI-K	2007	Westwood, Vegas	Congestion window increase steps as a function of the achievable rate and queuing delay estimates	S	Experimental			
TCP Africa [79]	VI-L	2005	HS-TCP, Vegas	Switching between fast (HS-TCP) and slow (NewReno) mode depending on the Vegas-type network state estimation	S	Experimental			ns2
Compound TCP [80]	VI-M	2005	HS-TCP, Vegas	Two components (slow and scalable) in the congestion window calculation	S	Experimental	Vista, S'08, XP*, S'03*	2.6.14–2.6.25*	
TCP Illinois [81]	VI-N	2006	NewReno, DUAL	Additive increase steps and multiplicative decrease factors as functions of the queuing delay	S	Experimental		>2.6.22	
YeAH TCP [82]	VI-O	2007	STCP, Vegas	Switching between fast (STCP) and slow (NewReno) mode depending on a combined Vegas-type and DUAL-type estimate, precautionary decongestion	S	Experimental		>2.6.22	

[1] TCP specification modification: S = the sender reactions, R = the receiver reactions, P = the protocol specification
[2] Microsoft™ operating systems: S for server versions [3] Network simulators * optional or available in patch form

increase coefficient α in *Congestion Avoidance* and decrease factor β after a minor loss detection (during the *Fast Recovery* phase) by functions of the congestion window size ($\alpha(w)$ and $\beta(w)$, respectively).

These functions $\alpha(w)$ and $\beta(w)$ are obtained based on the above-mentioned objectives defined in terms of the achievable congestion window size and the required loss rate (bold curve in Figure 40). That is, on the one hand, HS-TCP should be able to utilize the 10 Gbps link for the network with a loss rate not exceeding 10^{-7} (NewReno is unable to utilize this link if the loss rate exceeds 10^{-10}). On the other hand, it should act like a standard NewReno [13] in environments with loss probability higher than 10^{-3}.

The resulting functions $\alpha(w)$ and $\beta(w)$ vary from 1 and 0.5, respectively, when the congestion window is less than or equal to 38 packets (i.e., it has same behavior as NewReno when the congestion window is small) to (and beyond) 70 and

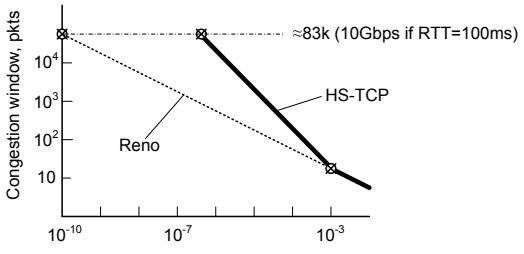


Fig. 40. Objective of HS-TCP

0.1 when the congestion window is more than 84k packets. Figure 41 shows the schematic comparison between HS-TCP and NewReno behavior during *Congestion Avoidance*/*Fast Recovery* phases. As we can see, at high congestion window sizes (or low loss rates) HS-TCP probes the network resources more aggressively than Reno and, at the same time, reacts more conservatively to loss detection events. This behavior considerably increases the efficiency of high-speed/long-delay networks. However, the tradeoff is an increased level of packet losses: more packets are lost during congestion events, which occur more frequently.

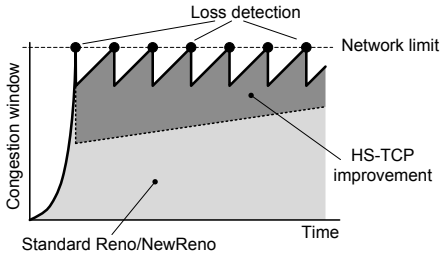


Fig. 41. Congestion window dynamics of HS-TCP

The high-speed/long-delay networks create one more problem for TCP. During the initial *Slow Start* phase when an approximate network limit is still unknown, the unbounded exponential probing (see Tahoe in Section II-A) can lead to a loss of extremely large numbers of packets. For example, in a 10 Gbps link with 100 ms RTT, *Slow Start* (in the worst case) can cause a loss of about 83,000 packets, which is approximately 120 MBytes of wasted network resources. To resolve this problem, Floyd [83] proposed a complementary algorithm that bounds the maximum increase step during *Slow Start* to 100 packets (*Limited Slow Start*). It is expected that this limitation will not have a significant impact on performance. There are two reasons why this is so. First, *Slow Start* operates only during initialization, or re-initialization after a timeout. In other words, it is insignificant for long-lived flow performance. Second, it takes about 8 seconds to fully utilize a 1 Gbps link with 100 ms RTT, which is assumed to be a reasonable payoff for a significant reduction of induced packet losses.

Another important question for a new congestion control algorithm is how flows that utilize it interact with each other (intra-fairness) and with other flows (inter-fairness), including standard TCP flows. By definition, during severe congestion situations (when the loss probability is high) HS-TCP is equivalent to standard Reno and thus inherits all its characteristics. In high-speed/long-delay networks, HS-TCP explicitly does not consider fairness to standard TCP flows to be significantly important, because standard flows cannot effectively utilize the available network resources. However, intra-fairness property is highly important. Fortunately, it can be shown that because HS-TCP does not change the core additive increase multiplicative decrease (AIMD) concept of NewReno—namely, that during *Congestion Avoidance* the window is increased by a constant number of packets each RTT and decreased during *Fast Recovery* by a fraction of itself (Figure II-A)—NewReno's intra-fairness properties are preserved. However, HS-TCP has substantial problems with fairness if flows have different RTTs. Although this problem is inherited from Reno [71], subsequent research discovered that AIMD coefficient scaling (functions instead of constants) significantly intensifies this problem. A number of congestion control algorithms discussed later in this survey (Hybla in Section VI-D, H-TCP in Section VI-C, FAST in Section VI-G, CUBIC in Section VI-F) address this problem and, at the same time, preserve data transfer effectiveness in high-speed/long-delay networks.

B. STCP

Kelly [66] proposed STCP (Scalable TCP) as an alternative to HS-TCP (Section VI-A) to solve the data transfer effectiveness problem in high-speed/long-delay networks. Instead of complicated AIMD coefficient calculations, STCP rejects the core AIMD concept and introduces a multiplicative increase multiplicative decrease idea (MIMD). In other words, during *Congestion Avoidance* an STCP flow increases its congestion window w by a fraction α of the window size with each RTT (i.e., $w = w + \alpha \times w$, where $\alpha = 0.01$). During *Fast Recovery*, it reduces the congestion window by a different fraction β upon detecting a loss (i.e., $w = w - \beta \times w$, where $\beta = 0.125$). A hypothetical congestion window dynamic of STCP looks similar to that of HS-TCP, but with increased frequency and sharpness of increase/decrease phases (Figure 42).

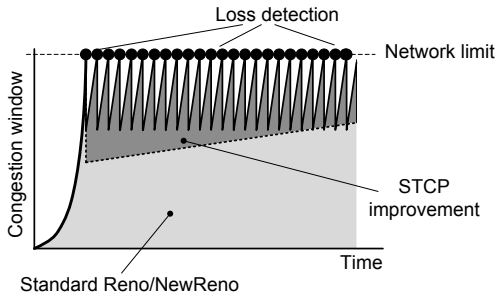


Fig. 42. Congestion window dynamics of STCP

Clearly, the proposed modifications resolve the target problem by making the increase/decrease dynamics follow exponential functions, which scale quite well in many environments. However, the solution creates a number of critical problems. First, from Figure 42 we can easily recognize that even one STCP flow moves the network to a state of nearly constant congestion. This is generally undesirable for most networks. Second, inter-fairness characteristics (i.e., fairness to standard flows) are similar to HS-TCP: in the low-loss rate zone ($<10^{-3}$), STCP does not even try to be inter-fair, assuming that standard TCP flows cannot effectively utilize network resources; in the high-loss zone, STCP behaves like standard TCP. Third, the MIMD approach does not conceptually provide intra-fairness (i.e., fairness between STCP flows). That is, under the assumption that two STCP flows are experiencing the same RTTs and are able to detect a packet loss simultaneously, the flow with the larger initial share will always have an advantage (Figure 43). This happens because

multiplicative increase and multiplicative decrease policies essentially preserve a ratio between congestion window sizes of the flows. Finally, it can be shown that due to MIMD policies, an STCP flow is extremely unfair, both to STCP and to standard TCP flows that have higher RTT values [71].

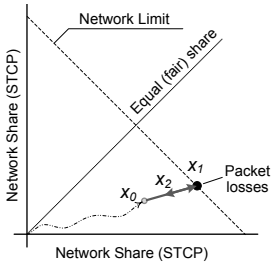


Fig. 43. Convergence diagram of the two STCP flow competition
$x_0 - x_1$, $x_2 - x_1$ multiplicative increase (a flow with the larger congestion window increases more than a flow with the smaller)
$x_1 - x_2$ multiplicative decrease (a flow with the larger congestion window decreases more than a flow with the smaller)

C. H-TCP

Leith and Shorten [68], [69] presented one more alternative to congestion control for TCP, called H-TCP (Hamilton TCP), which is intended to have good fairness (inter-, intra-, RTT-) and effectiveness properties. The key idea of their proposal is that the congestion window increase step α in *Congestion Avoidance* should be a non-decreasing function of the time elapsed since the last congestion event (Δ). In one sense, this is similar to HS-TCP (Section VI-A) where the network resource probing steps (i.e., the congestion window increase per RTT) grow as the congestion window itself is growing. However, the functional dependence of the elapsed time Δ has one significant advantage over the dependence of the congestion window: namely, that no matter how large the initial congestion window sizes have been, flows experiencing the same network conditions will exhibit the same congestion window increase dynamics. In other words, an H-TCP flow is fair to other H-TCP flows present in the same network path. To demonstrate that this holds, one can build a convergence diagram of the two competing H-TCP flows and observe that it looks similar to Figure II-A, assuming that after a loss detection, both flows decrease their congestion windows by half.

More specifically, H-TCP defines the increase in the congestion window w as $\alpha(\Delta)$ for each RTT (equivalent to increase as a fraction $\alpha(\Delta)/w$ for each reception of non-duplicate ACK, where w is the current congestion window size. $\alpha(\Delta)$ is the polynomial function over time Δ elapsed since the last congestion event, as follows:

$$\alpha(\Delta) = 1 + 10(\Delta - \Delta_{low}) + 0.5 \cdot (\Delta - \Delta_{low})^2$$

where Δ_{low} is a predefined threshold of H-TCP's compatibility mode—i.e., whenever $\Delta < \Delta_{low}$, $\alpha(\Delta) = 1$.

It can be noted that this definition of $\alpha(\Delta)$ still leads to some degree of RTT-unfairness. For example, let us consider two H-TCP flows competing with each other and having different RTT values (Figure 44). If we assume that $\alpha(\Delta)$ is calculated once per RTT at time 0, T_1, and T_2 (note, with this assumption we do not change the H-TCP principle, but it allows us to highlight the problem), we see that a flow having a longer RTT always loses to a flow with a shorter RTT. To mitigate this effect, H-TCP defines an optional mechanism of scaling the $\alpha(\Delta)$ to a reference RTT (RTT_{ref}) which, as an example, can be 100 ms: $\alpha'(\Delta) = \alpha(\Delta) \times RTT/RTT_{ref}$.

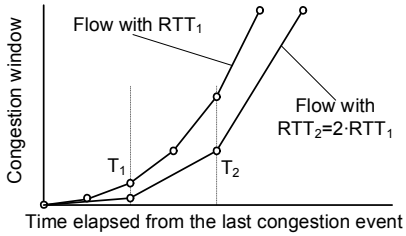


Fig. 44. Rationale of H-TCP's RTT-unfairness

In addition to these changes in the *Congestion Avoidance* phase, the H-TCP proposal includes a small modification of the congestion window reduction policy in *Fast Recovery*. More specifically, upon detecting a packet loss, H-TCP estimates the achieved flow's throughput $B(k)$ and compares it with the estimate of the preceding loss event $B(k-1)$. If the absolute value of the relation $[B(k) - B(k-1)]/B(k-1)$ is less than 0.2, the congestion window is reduced by the ratio RTT_{min}/RTT_{max}; otherwise, the coefficient 0.5 is used. However, later in the Internet-Draft proposal, Leith [69] removed the *Fast Recovery* modification from H-TCP.

D. TCP Hybla

Caini and Firrincieli [70] emphasized the problem of degradation of TCP throughput with standard congestion control—NewReno—in long-delay networks. It can be shown that in NewReno's *Congestion Avoidance*, the congestion window size w is inversely dependent on RTT, and the the TCP throughput B has an upper bound that is inversely dependant on RTT^2 (the throughput can be approximated by the expression $B \cong w/RTT$). Clearly, a flow with a shorter RTT will always have an advantage compared to a flow with a longer RTT. In heterogeneous networks, especially with satellite segments, the RTTs may be different by several orders of magnitude, potentially resulting in catastrophic unfairness in the network resource distribution.

To resolve this RTT-unfairness problem, a Hybla algorithm has been proposed [70]. This algorithm introduces modifications to the NewReno's *Slow Start* and *Congestion Avoidance* phases that make them semi-independent of RTT. In particular, to obtain the normalized increase steps in both phases, the scaling factor ρ is calculated according to the equation $\rho = RTT/RTT_{ref}$, where RTT_{ref} is a reference RTT (e.g., 25 ms). Formally, the increase steps upon receiving ACK packet are defined as follows:

$$w = w + 2^{\rho} - 1, \quad \text{in } \textit{Slow Start}$$
$$w = w + \rho^2/w, \quad \text{in } \textit{Congestion Avoidance}$$

This definition is illustrated in Figure 45, where three flows having different RTT values are presented. The higher the RTT value, the higher the ratio ρ becomes and the congestion window is increased more rapidly with each ACK packet reception. As a consequence, during the same time period, flows will result in different congestion window values. However, if we calculate the upper bound of the TCP throughput (i.e., the ratio between the congestion window and the RTT), we will see that all three flows can transmit data at similar rates ($\approx$ 2 MByte/s after 500 ms).

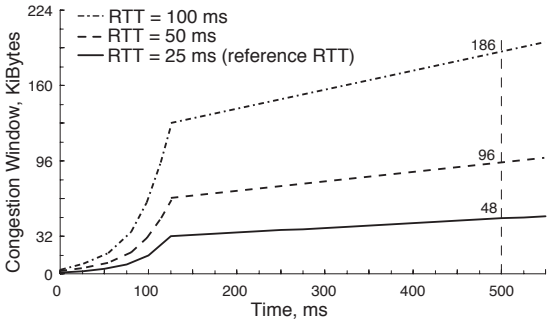


Fig. 45. Congestion window evolution in Hybla

In addition, Hybla introduces two more techniques that complement the congestion control: pacing the transmission of data packets [84] and estimating the initial slow-start threshold using the packet pair algorithm [76]. The pacing is essentially setting up a minimal delay between transmission of any two consecutive packets. It is meant to smooth the burst-nature of TCP transmissions. The packet pair algorithm provides an ability to estimate the network path capacity. Knowledge of the network capacity may help us improve the convergence speed and, to some degree, provides a scalability in high-BDP networks.

A number of experimental evaluations [70] have confirmed remarkable RTT-friendliness of the Hybla algorithm. However, the cost of this friendliness is an increased aggressiveness of the flows with larger RTT values. At the same time, these flows have a slower feedback rate—a packet loss can be detected no earlier than delivery of a packet can be confirmed (i.e., feedback rate is proportional to 1/RTT). Thus, more aggressive flows can easily congest the network before they detect any packet loss. To some extent, pacing technique soften, but cannot eliminate this problem. Additionally, Hybla is designed to fall back to the standard mode (to Reno-like congestion control rules) if a flow's RTT is less than a predefined reference value. This property limits applicability of Hybla to satellite-like channels: Hybla, similar to the standard Reno, is unable to work effectively in high-speed networks with relatively small delays.

E. BIC TCP

Xu et al. [71] pointed out the RTT unfairness problem of HS-TCP (Section VI-A) and STCP (Section VI-B). For example, if we assume that two competing flows can detect a loss simultaneously (a synchronized loss detection), the analytical calculations reveal that an HS-TCP flow having an RTT x times smaller will get a network share which is $x^{4.56}$ times larger. Similar calculations for STCP show that, in theory, the STCP flow with the smaller RTT will always get all of the network resources, and a flow with the higher RTT will get nothing (i.e., absolute unfairness). The problem in both cases lies in the way these algorithms discover network resources: a flow with a larger congestion window will try to increase its share more than a flow with a smaller window.

In an attempt to create a congestion control that can scale well in any high-BDP (high bandwidth-delay product) network and yet to remain relatively RTT-fair, Xu et al. [71] proposed a BIC (Binary Increase Congestion control) algorithm. This algorithm extends NewReno with an additional operational phase, *Rapid Convergence*. This phase rapidly discovers, in a binary search manner, the optimal congestion window size (i.e., the value corresponding to the available network resources) by relying on detection of a packet loss as an indication of congestion window overshooting. Schematically, the congestion control concept of BIC as a search problem is illustrated in Figure 46. While the network successfully delivers data packets (i.e., the sender receives all ACKs during the last RTT), the congestion window is updated to the median of the search range between minimum w_{min} and maximum w_{max} congestion window sizes (initially, w_{min} is set to one and w_{max} to some arbitrary high value). Besides updating the congestion window, an indication of successful data delivery raises the lower boundary w_{min} to the previous congestion window size—the value when the network is expected to be congestion-free. As soon as packet loss is detected (e.g., three duplicate ACKs are received), BIC sets the upper search boundary w_{max} to the current congestion window size—the value when the network is experiencing congestion—and enters the well-known *Fast Recovery* phase, similar to NewReno (see Section II-D). Additionally, to increase the convergence rate in the low-loss network environments, BIC reduces the multiplicative decrease coefficient from 0.5 to 0.125 (i.e., $w = w - 0.125 \cdot w$) when the congestion window size is more than 38. This number is borrowed from HS-TCP and is aimed at providing compatibility to Reno in environments with loss rates exceeding 10^{-3}.

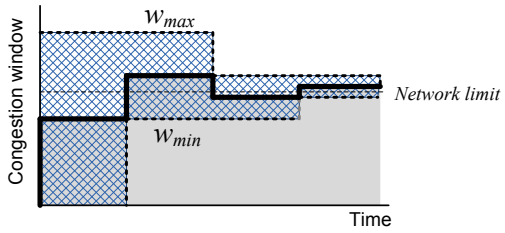


Fig. 46. Binary search for the optimal congestion window in TCP BIC

Though a true binary search algorithm features a very fast (logarithmic) convergence time, in a high-BDP network it may create the same problem that was discovered in Slow-Start: if the congestion window is increased too fast, a large number of packets can be lost (see Section VI-A). For this reason, BIC not only adopts HS-TCP's *Limited Slow Start*, but it also limits the increase in *Rapid Convergence* when the

search range is too wide. In other words, during the RTT, *Rapid Convergence* is not allowed to increase the congestion window by more than some predefined value S_{max}. To address the opposite case when the search range is too narrow—near the estimated optimum—BIC defines the congestion window increase by at least some constant S_{min} number of packets. Finally, when the current congestion window value during *Rapid Convergence* becomes very close to, or exceeds, the target congestion window value, BIC enables the *Limited Slow Start* phase with an unlimited slow-start threshold value. This action is to discover a new upper bound and restart the binary search (Figure 47).

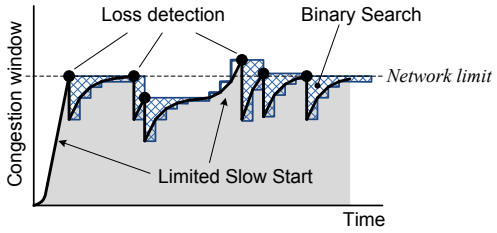


Fig. 47. Congestion window dynamics in TCP BIC

The BIC approach for optimal congestion window discovery has a unique feature for loss-based congestion control approaches—the congestion window probing steps decrease as the window approaches a target value. Xu et al. [71] showed that in a synchronized loss model, the congestion window ratio of two flows with different RTTs (RTT-fairness) changes from $e^{(1/RTT_1 - 1/RTT_2)t \cdot \ln 2}$ for small window sizes to $\frac{RTT_1}{RTT_2}$ for large ones. In other words, in theory BIC is no less RTT-fair than the standard Reno algorithm. However, a number of later experimental evaluations [85], [72] showed that in certain environments BIC may have low RTT-fairness and inter-fairness values (fairness to other deployed TCP congestion controls). A revised version of BIC, called CUBIC [72], is meant to improve these properties and is discussed in Section VI-F.

F. CUBIC TCP

Rhee and Xu [72] noted the highly challenging problem of creating a simple congestion control algorithm that scales well in high-BDP networks, and at the same time has good intra-, inter-, and RTT-fairness properties. Some of the previous congestion control proposals (e.g., HS-TCP, STCP, BIC) enforce inter-fairness (i.e., fairness to the standard TCP flows) by switching to standard congestion window update rules in high-loss environments. The switching criteria in most cases is a pre-calculated congestion window size w which corresponds to a certain loss rate p ($w = 1.2/\sqrt{p}$). For example, the threshold in HS-TCP is set to 38 packets, which corresponds to a loss of one out of every 1000 consecutive packets ($p = 10^{-3}$ pkt/s). However, this definition of loss rate is not the ideal guideline, especially in heterogeneous networks where RTT can vary significantly. For example, in a network with 10 ms RTT, the loss rate 10^{-3} allows one loss every 380 ms, while in a network with 100 ms RTT, the same rate allows only one loss every 3.8 seconds. Thus if two flows competing in the same bottleneck link have different RTTs, the flow with the larger RTT is likely to remain in a compatible mode all the time, while the flow with the smaller RTT quickly switches to a scalable mode and acquires all available resources. This observation allowed Rhee and Xu [72] to propose CUBIC congestion control, which enhances the previously introduced BIC algorithm (Section VI-E) with RTT-independent congestion window growth functions. To accomplish this, CUBIC borrows the H-TCP approach (Section VI-C) of defining the congestion window w as a cubic function of elapsed time Δ since the last congestion event, as follows:

$$w = C\left(\Delta - \sqrt[3]{\beta \cdot w_{max}/C}\right)^3 + w_{max}$$

where C is a predefined constant, β is a coefficient of multiplicative decrease in *Fast Recovery*, and w_{max} is the congestion window size just before the last registered loss detection. This function preserves not only RTT-fairness, since the window growth does not depend much on RTT, but also scalability and intra-fairness properties of BIC's *Limited Slow Start* and *Rapid Convergence* phases. The function has a very fast growth when the current window w is far from the estimated target w_{max}, and it is very conservative when w is close to w_{max}. Figure 48 shows the theoretical dynamics of the growth of the congestion window in CUBIC. In the initial step labeled *1*, the target window w_{max} is unknown and is discovered using the right branch of the cubic function. This discovery is much more conservative than the exponential discovery used in conventional *Slow Start*, but it is still scalable to high-BDP networks. At later stages, following a reduction upon detecting a loss, the congestion window gently approaches the target (phase *2*). If a loss is detected before w_{max} is reached, the target is updated. If it was a temporary congestion event, we will see a congestion window growth according to both left and right branches of the cubic function (phase *3*).

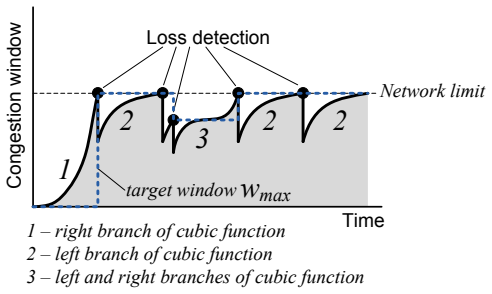


Fig. 48. Congestion window dynamics in CUBIC

Additionally, CUBIC provides a mechanism to ensure that its performance is no worse than the performance of the standard (Reno) congestion control. This mechanism includes calculating a supplementary congestion window size w_{reno} that approximates the performance of a corresponding standard Reno flow. Because the congestion window in CUBIC can be reduced by a fraction β different from 0.5 (i.e., generally $\beta_{cubic} \neq \beta_{reno}$), the appropriate performance (an average sending rate) can be achieved only if the supplementary congestion window increase steps are scaled with $s = 3 \times (\beta -$

$1)/(\beta+1)$ [72]. Formally, this can be written as an increase of the window w_{reno} by s every RTT. If CUBIC detects that the supplementary window w_{reno} exceeds the main window, the latter is reset to be equal to the former.

The good performance and fairness properties of CUBIC were confirmed by various experimental studies [72], [82] and by real-world measurements. CUBIC is currently the second most-used congestion control algorithm for TCP, due to the fact that it has been the default for the Linux TCP suite since 2006 (i.e., Linux kernel version 2.6.16). Nevertheless, CUBIC does not have 100% network resource utilization and can induce a large number of packet losses in the network (as long as a loss is the only signaling mechanism).

G. FAST TCP

Jin et al. [73], [74], [75], inspired by the Vegas idea of congestion control with the queuing delay as a primary congestion indicator (see Section II-G), introduced a FAST algorithm. In some sense, FAST may be considered a scalable variant of Vegas that defines a periodic congestion window update based on the internal delay-based estimate of the network state. However, there are two fundamental differences between Vegas and FAST: FAST defines a periodic fixed-rate congestion window update (e.g., each 20 ms) and, to calculate the new target congestion window size, FAST uses a specially designed equation which incorporates a simple delay-based congestion estimation feature:

$$w = w \cdot \frac{RTT_{min}}{RTT} + \alpha$$

where w is a current congestion window size, RTT and RTT_{min} are current and minimum RTT, and α is an important protocol parameter, as described below.

According to this equation, if the network is experiencing congestion ($RTT > RTT_{min}$), FAST will decrease the congestion window (use of the network resources) proportionally to the congestion level estimated using RTT measurements (RTT_{min}/RTT); otherwise, the window will be increased based solely on the predefined parameter α. Selection of α has conflicting effects on two important protocol parameters: scalability and stability. In other words, if α is too large, the protocol will scale easily to any high-BDP (high bandwidth-delay product) networks, but it will have substantial convergence problems (the stable state when $w = w \times RTT_{min}/RTT + \alpha$ will be barely reachable). In the opposite case, when α is too small, FAST will easily stabilize but will have scalability problems (e.g., if $\alpha = 1$, FAST behavior is practically equivalent to Vegas). Although the problem of accurate selection of α is still an open issue, FAST's authors have concluded that α should be a constant. Attempts at making α vary depending on the congestion window size, and RTT measurements are reported to lead to substantial intra- and inter-unfairness [75].

To make the equation-based algorithm tolerant of short-term fluctuations in network parameters, FAST uses a well-known technique of exponential smoothing of the calculated congestion window value. In addition, FAST limits a potential increase in the congestion window (when $\alpha \gg w$) to be no more than a current value, which is roughly equivalent to the increase in the standard Slow-Start mode. The only difference is that FAST increases the congestion window based on the internal timer expiration (e.g., each 20 ms), but Slow-Start is clocked by ACK packets reception.

Although simulation-based and real-world experiments show remarkable intra-fairness, RTT fairness, stability, and scalability, ongoing research [75], [86] recognizes a number of serious issues with the design. First, FAST's characteristics depend highly on the true minimal RTT value, which is hard to calculate in some environments (e.g., when routes tend to be dynamic) without relying on additional messaging from the network. Second, the RTT is not always a good substitute for the queuing delay, especially when there is congestion along the reverse path or when there are route changes. Finally, the proposed congestion window update rule is not friendly to standard TCP (Reno, NewReno, or SACK), even in small-BDP networks.

H. TCP Libra

Marfia et al. [76] proposed Libra as another variant of congestion control to resolve the scalability issues in standard TCP, while preserving and improving the RTT-fairness properties. Libra's design is based on NewReno (Section II-D) and modifies the *Congestion Avoidance* congestion window increase steps to follow a specially designed function of both the RTT and the bottleneck link capacity. The latter value in Libra is estimated using a well-known packet pair technique [87]. Formally, in Libra's *Congestion Avoidance*, if no loss has occurred, the congestion window is increased by α packets every RTT, according to the equation:

$$\alpha = k_1 \cdot C \cdot P \cdot RTT^2/(RTT + \gamma)$$

where RTT is the current RTT estimate, γ and k_1 are predefined constants (e.g., 1 and 2, respectively), C is a value responsible for Libra's scalability and represents the capacity of the bottleneck link estimated using the packet pair technique, and P is a penalizing factor which reduces the increase step if the network experiences congestion (e.g., the congestion window size is close to the convergence point). In particular, penalizing factor P can be represented with the expression based on queuing delay measurements:

$$P = e^{-k_2 \times Q/Q_{max}}$$

where k_2 is some constant (e.g., 2), and Q and Q_{max} are current and maximum queuing delay estimates (see Section II-B).

The rationale of the congestion window increase steps α is as follows: the first part of the functional dependence $k_1 \cdot C$ makes the increase steps scalable to the bottleneck link capacity. The penalizing part P forces Libra to decrease the network resource probing intensity (the congestion window increase steps) exponentially, if the estimated level of buffering in the network (Q/Q_{max}) increases. The last part, $RTT^2/(RTT + \gamma)$, is responsible for Libra's RTT-fairness. If RTT is significantly less than the constant γ, the increase steps are scaled to RTT^2—the essential requirement for RTT-fariness (see Section VI-D). The constant γ is selected in a way that considers all links with an RTT close to or more than

γ to have some pathological problems when RTT-fairness is not an issue.

In addition, Libra also defines a change in the multiplicative decrease policy of the *Fast Recovery* phase ($w = w - \beta \times w$): the decrease coefficient β scales with the expression $\theta/(RTT + \gamma)$, where θ and γ are constants. Although this scaling factor is derived analytically [76], the recommended values for these constants (θ and γ are equal to 1 second) make the scaling factor close to 1 in most cases. Thus, it is not very significant. Moreover, when the current RTT value is large (e.g., potential congestion in the network), the scaling factor will reduce β further. Yet this is the opposite of what congestion control should do: the decrease should be maximized in the presence of congestion and minimized when the network is in a congestion-free state.

A number of experimental evaluations (ns2 simulations) show that Libra can help improve the high-BDP link utilization and fairness properties of TCP. However, the same results show that Libra does not always outperform other congestion control approaches, including the non-scalable Reno with selective ACKs (Section II-E). Additionally, due to high reliance on the queuing delay estimation (i.e., RTT_{min}, RTT_{max}, and RTT measurement consistency), Libra's properties, similar to FAST (Section VI-G) and C-TCP (Section VI-M), will further worsen because of estimation biases.

I. TCP New Vegas

Sing and Soh [25] recognized the advantages of the delay-based congestion control approach presented in Vegas (Section II-G). However, they also found that it has three serious problems (two of them have been inherited from Reno, Section II-C): (a) it cannot effectively utilize high-BDP links, (b) during the (re-)initialization phases (*Slow Start* and *Fast Recovery*) the Vegas congestion control can generate very bursty traffic, and (c) Vegas' estimation of network buffering can be significantly biased if receivers use the standardized delayed ACK technique [21], [11].

To reduce the convergence time and improve to some degree the high-BDP link utilization, the proposed New Vegas algorithm defines a new phase called *Rapid Window Convergence*. The key idea of this phase is not to immediately terminate the *Slow Start* phase when the estimate of network buffering exceeds the threshold ($\Delta > \alpha$), but to continue the opportunistic exponential-like resource probing with reduced intensity. In detail, when New Vegas's *Slow Start* detects that the threshold has been exceeded (early congestion detection), it remembers the current congestion window value in a special state variable w_r and switches to the *Rapid Window Convergence*. In this state, for every RTT, the congestion window is allowed to be increased by x packets:

$$x = (w_r)^{-2^{3+n}}$$

where n is a number of times the early congestion indicator triggers in the *Rapid Window Convergence* phase. According to the proposal [25], when n becomes more than 3, *Rapid Window Convergence* terminates and normal Vegas-like *Congestion Avoidance* takes its place. At any point, if a packet loss is detected, NewVegas reacts exactly the same as the original Vegas algorithms. In other words, if the loss is detected using three duplicate ACKs, NewVegas switches to *Fast Recovery* followed by *Congestion Avoidance*; if the loss is detected using the RTO, NewVegas resets the congestion window size and moves to *Slow Start*.

In order to solve the second problem of generation of bursty traffic during initialization and re-initialization, NewVegas applies the well-known packet pacing technique; it sets-up a minimal delay between transmission of any two consecutive packets [62], [88]. Although it is reported to have a negative impact on TCP Reno performance [84], NewVegas authors believe and experimentally confirm that with delay-based congestion controls, packet pacing has only a positive effect.

The last problem of the estimation bias is solved by requiring the sender to transmit data packets in pairs. In terms of TCP, this means that if there are data to be sent and the current value of the congestion window allows sending only one data packet, NewVegas will hold any transmission until the window increases by at least one packet. Clearly, this technique can help overcome the RTT estimation problem, when the TCP receiver does not immediately respond to each data packet, but waits for a timeout or another data packet. However, this is based on the assumption that two data packets will not be separated much during delivery (e.g., due to congestion) and that the TCP receiver sends ACK packets for every other data packet, as a minimum. Neither of these assumptions is entirely true in real networks. Thus, packet pairing has questionable benefits on the precision of RTT-based estimations (e.g., queuing delay). Moreover, the RTT measurement can be improved significantly just by employing the Timestamp option [24].

Although NewVegas authors have recognized the problem of high-BDP link utilization, the proposed *Rapid Window Convergence* resolves only one part of the problem: this phase of scalable congestion window increase steps intends only to improve the early termination of *Slow Start*. No scalable features are designed for *Congestion Avoidance* and *Fast Recovery*, which limits the potential applicability of NewVegas in the future.

J. TCP-AR

Shimonishi and Murase [77] presented TCP-AR (Adaptive Reno) as another approach to improve TCP performance and preserve friendliness to standard TCP in high-speed networks. It extends TCPW-BBE (Section V-E) with a scalable congestion window probing in the *Congestion Avoidance* phase. More specifically, the congestion window increase function is defined to have two components: a slow constant increase component W_{base} (increased by one for every RTT) and a scalable increase component W_{probe} (increased by a function of the Westwood-like achievable rate estimate and the queuing delay, see Sections II-B and V-A respectively). The scalable component is a continuous function that has two important properties. First, when the network is congestion-free (i.e., when queuing delay is close to zero), the function gives a value close to the Westwood-like achievable rate estimate. Second,

when the network is experiencing congestion (i.e., when queuing delay is near maximum), the value of the scalable component W_{probe} is zero. Figure 49 shows conceptually the congestion window dynamics of the TCP-AR algorithm.

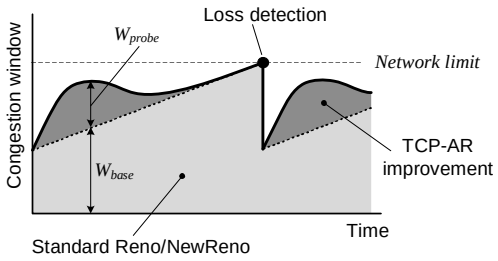


Fig. 49. Congestion window dynamics in TCP-AR

Experimental results showed that TCP-AR can improve the network utilization successfully and at the same time preserve a good level of intra-fairness. However, relying on the queuing delay and achievable rate metrics makes this algorithm vulnerable if RTT measurements should become noisy. In the worst case, when the queuing delay is wrongly estimated to be close to the maximum, TCP-AR totally loses its ability to scale in high-BDP networks.

K. TCP Fusion

Kaneko et al. [78] presented a Fusion algorithm which, in a fashion similar to TCP-AR (Section VI-J), combines the ideas of Westwood's achievable rate (Section V-A), DUAL's queuing delay (Section II-G), and Vegas' used network buffering (Section II-G) estimations. Instead of TCP-AR's congestion window increase in *Congestion Avoidance* as a continuous function over the queuing delay, Fusion defines three separate linear functions which are switched, depending on an absolute (i.e., expressed in seconds) queuing delay threshold value. If the current queuing delay is less than the predefined threshold (zone 1 in Figure 50), the congestion window is increased at a fast rate each RTT by a predefined fraction of Westwood's achievable rate estimate (scalable increase). If the queuing delay grows more than three times the threshold (zone 3 in Figure 50), the congestion window decreases by the number of packets buffered in the network (i.e., the Vegas estimate). In the case where the queuing delay lies somewhere in the range between one and three times the threshold (zone 2 in Figure 50), the congestion window remains unchanged. To make Fusion behave at least as well as the standard Reno congestion control, a conventional Reno-like window w_r is maintained along with the Fusion congestion window w_f. If w_f becomes smaller than w_r, the w_f is reset equal to w_r.

In addition, Fusion changes the constant congestion window reduction ratio β in *Fast Recovery* to the value $\beta = max(0.5, RTT_{min}/RTT)$. This ratio is essentially a simplified form of Westwood's, where the sampling interval equals the RTT [78].

Although experimental results of evaluating Fusion [78] have shown some improvements in terms of utilization and fairness characteristics in comparison to other scalable algorithms (e.g., C-TCP, HS-TCP, BIC, FAST), Fusion not only has

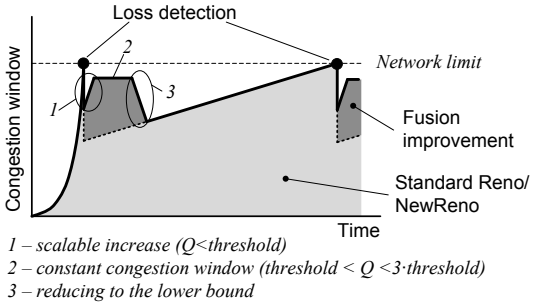


Fig. 50. Congestion window dynamics in Fusion

the same vulnerabilities as TCP-AR, but also introduces a new, more serious problem. Defining the threshold in absolute terms requires manually adapting Fusion to a particular environment. This manual configuration is highly undesirable and usually impossible to perform. Another problem of Fusion is the way it quickly defaults to standard congestion window control rules. As one can see in Figure 50, in certain cases Fusion may stay in the compatible, slow, non-scale mode most of the time.

L. TCP Africa

King et al. [79] were concerned with the problems of several previously proposed congestion control algorithms for high-BDP networks, including HS-TCP (Section VI-A) and STCP (Section VI-B). In response to these concerns they developed the Africa (Adaptive and Fair Rapid Increase Congestion Avoidance) algorithm. This algorithm combines the aggressiveness (scalability) of HS-TCP when the network is determined to be congestion-free and the conservative character of standard NewReno (Section II-D) when the network is experiencing congestion. The congestion/non-congestion criteria was borrowed from the Vegas algorithm (see Section II-G): the estimate of network buffering Δ is compared to some predefined constant α. More formally, if Africa sees that there is little buffering ($\Delta < \alpha$), it moves to *fast mode* and directly applies the HS-TCP rules of the *Congestion Avoidance* and *Fast Recovery* phases. These dictate congestion window increase and decrease steps as functions of the congestion window itself (see Figure 51). Otherwise, it moves to *slow mode* and applies the Reno rules: increase by one, decrease by half.

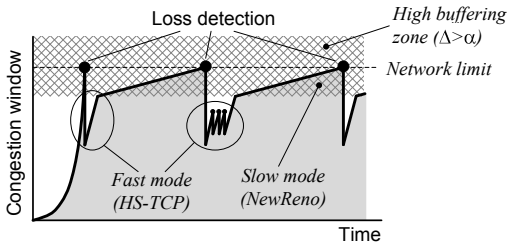


Fig. 51. Congestion window dynamics of TCP Africa

In a number of simulations conducted by its authors, Africa showed good network utilization in high-BDP networks, lower induced loss rate compared to HS-TCP and STCP, and fairness properties (intra-, inter-, RTT-) comparable to those exhibited

by NewReno flows. Unfortunately, Africa has not been implemented and evaluated in real networks. However, the idea of multiple-mode congestion control for high-BDP networks with delay-based mode-switching has been widely adopted by several proposals discussed later. For example, the dual-mode C-TCP algorithm (Section VI-M) is currently the most deployed TCP congestion control in the world, since it is embedded into the Microsoft Windows operating system (see Table V).

M. C-TCP

Tan et al. [80] presented C-TCP (Compound TCP), a congestion control approach similar in spirit to Africa (Section VI-L). It also tries to use a delay-based estimate of the network state to combine the conventional Reno-type congestion control (Section II-C) with a congestion control that is scalable in high-BDP networks. However, instead of explicitly defining the fast and slow modes, C-TCP defines an additional scalable component w_{fast} to be added to the final congestion window calculations ($w = w_{reno} + w_{fast}$). This component is updated according to the slightly modified HS-TCP rules (Section VI-A) but only when the Vegas estimate Δ (Section II-G) shows a small level of network buffering ($\Delta < \alpha$, where α is some small predefined constant). When the estimate exceeds the threshold α, the scalable component w_{fast} is gently reduced by a value proportional to the estimate itself ($w_{fast} = w_{fast} - \zeta \cdot \Delta$, where ζ is a predefined constant). This reduction can be understood as a smooth transition between scalable HS-TCP and slow Reno modes, as opposed to the instant transitions between the fast and slow modes of Africa. As a result, the theoretical congestion window dynamics of C-TCP (Figure 52) are very similar to Africa's with the exception that after the threshold has been exceeded ($\Delta > \alpha$), we will see a convex curve (shown in hatched bar) in the transition from the scalable HS-TCP to the slow Reno mode.

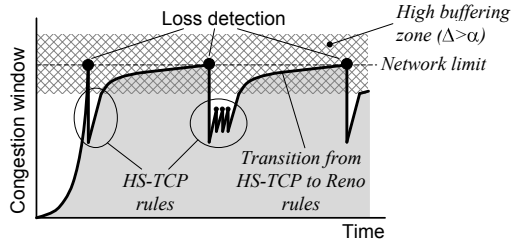


Fig. 52. Congestion window dynamics of C-TCP

Both simulation results and real-world performance evaluation show substantial advantages of the C-TCP scheme: a good utilization of the high-BDP links and good intra-, inter-, and RTT-fairness properties. As a result, C-TCP has replaced the conventional congestion control for TCP in Microsoft Windows operating systems and is currently the most deployed congestion control worldwide. However, on the weaker side, because C-TCP relies on the Vegas estimate, it has inherited the Vegas sensitivity to the correctness of RTT measurements. For example, if flows competing with each other in the same network observe different minimal RTT values (e.g., one flow already sending data when a second one appears), the flow seeing a higher RTT (which is equivalent to having a higher threshold α value) will be much more aggressive and unfair to the other flow.

N. TCP Illinois

Liu et al. [81] noted that congestion control algorithms that interpret a delay as a primary signal for inferring the network state (e.g., Vegas and Fast) are able to achieve a better efficiency and do not stress the network excessively, compared to congestion controls that rely only on packet losses (e.g., Reno, HS-TCP, STCP, etc.). However, the performance of delay-based algorithms may suffer greatly when the delay (RTT) measurements are very noisy, for example, due to a high volume of cross traffic, route dynamics, etc. To resolve this contradiction, the Illinois algorithm has been proposed. This algorithm, similar to Africa (Section VI-L) and C-TCP (Section VI-M), is based on NewReno (Section II-D) and is designed on the one hand to be very aggressive when the network is determined to be in a congestion-free state and on the other hand be very gentle when the network is experiencing congestion. However, Illinois has several implementation differences. It defines both the congestion window w increase steps α in *Congestion Avoidance* (i.e., $w = w + \alpha$ every RTT) and the decrease ratio β in *Fast Recovery* (i.e., $w = w - \beta \cdot w$, upon detecting loss using three duplicate ACKs) to be special functions of the queuing delay. The queuing delay calculation follows the definition introduced in DUAL (Section II-B). The increase coefficient α depends inversely on the queuing delay, while the decrease coefficient is directly proportional (Figure 53). The minimum and maximum values of α and β, and the queuing delay thresholds Q_1, Q_2, and Q_3, can be varied to achieve desired performance characteristics. In the Linux implementation, Illinois sets the default values of $\alpha_{max} = 10$, $\alpha_{min} = 0.3$, $\beta_{min} = 0.125$, $b_{max} = 0.5$, $Q_1 = 0.01 \cdot Q_{max}$, $Q_2 = 0.1 \cdot Q_{max}$, and $Q_3 = 0.8 \cdot Q_{max}$, where Q_{max} is a maximum queuing delay observed over the lifetime of the connection.

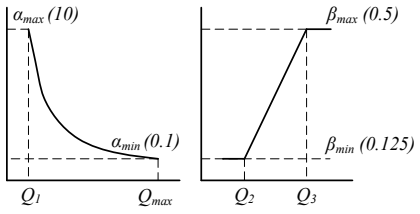


Fig. 53. Additive increase α and multiplicative decrease δ coefficients as a function of queuing delay Q

According to the Illinois specification, the α and β coefficients are updated once every RTT. However, to mitigate the effects of queuing delay measurement noise, the α coefficient is allowed to be set to the maximum, only if during several consecutive RTTs (e.g., 5) the value of the queuing delay is less than the first threshold Q_1. Additionally, Illinois switches to the compatibility mode ($\alpha = 1$ and $\beta = 0.5$) when

the congestion window size is less than a predefined threshold w_t (e.g., ten packets). This switch, similar to HS-TCP (Section VI-A) and STCP (Section VI-B), improves fairness properties of Illinois to some extent, making it behave like NewReno during severe congestion events. Figure 54 shows the key cases of the Illinois theoretical congestion window dynamics.

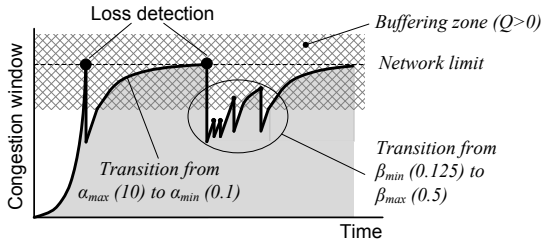


Fig. 54. Congestion window dynamics of TCP Illinois

The theoretical and experimental evaluation of Illinois showed that it is able to use the available resources in the high-BDP networks better than the standard Reno congestion control. At the same time, it preserves and improves the intra-, inter-, and RTT-fairness properties. However, although the queuing delay is a secondary parameter by which to infer the network state—i.e., it controls only the amount of the congestion window increase and cannot enforce its reduction—the advantages of Illinois can easily be nullified. It can fall back to the Reno mode ($\alpha = \alpha_{min}$ and $\beta = \beta_{max}$) whenever either the minimum or the maximum RTT values are incorrectly estimated or the RTT includes large random components (e.g., processing delay, different propagation delays when path frequently changes, etc).

O. YeAH TCP

Baiocchi et al. [82] introduced one more alternative for congestion control that combines packet loss detection and measurement of RTT as mechanisms to estimate the network state. Similar to Africa (Section VI-L), the proposed YeAH (Yet Another High-speed) algorithm defines the slow NewReno (Section II-C) and the fast STCP (Section VI-B) modes in *Congestion Avoidance* and *Fast Recovery* explicitly. For the former, the congestion window increases by at most one packet every RTT and decreases by half upon detecting a loss from three duplicate ACKs. In the latter, the congestion window is updated aggressively—increased by a fraction of the congestion window itself each RTT and decreased by another fraction which is much smaller than that in slow mode (i.e., less than half). To provide a reliable mechanism for mode switching, YeAH defines simultaneous use of two delay-based metrics: the Vegas-type estimate of a number of packets buffered in the network (see Section II-G) and the DUAL-type network congestion level estimate (see Section II-B). However, there are two differences in the definition of the latter metric from what was introduced in DUAL. First, in queuing delay calculations YeAH uses the minimum of recently measured RTTs (e.g., during the last RTT) instead of an averaged RTT. Second, the congestion level is measured, not as a fraction of the maximum queuing delay (e.g., Q/Q_{max}), but as a fraction of the minimum RTT observed during the connection lifetime. To summarize, if YeAH estimates a low level of packet buffering in the network ($\Delta < \alpha$, where α is a predefined threshold) and the queuing delay estimate shows a low congestion level ($Q/RTT_{min} < \varphi$, where φ is another predefined threshold), then it behaves exactly as STCP; otherwise, the slow Reno-like mode is enforced.

In addition to mode switching, YeAH includes two more mechanisms for improving robustness during congestion events and enhancing intra-fairness properties. The first mechanism, precautionary decongestion, is close in spirit to C-TCP (Section VI-M), whereby it reduces the congestion window w by the number of packets Δ estimated to be buffered in the network if this number exceeds a predefined threshold ε (i.e., $w = w - \varepsilon \cdot \Delta$). The second mechanism repeats another idea presented in C-TCP: the congestion window is restricted to be a value that is greater than if only Reno rules are applied. For this purpose, YeAH maintains a reference congestion window size w_{reno} that varies according to Reno rules. YeAH furthermore disables the precautionary decongestion if the reference window is more than the actual congestion window size.

Experimental evaluation showed that YeAH maintains high efficiency in high-BDP networks that maintain a network buffering at a very low level. Additionally, the results confirmed that approaches which combine delay-based and loss-based metrics (e.g., Africa, C-TCP, YeAH) can improve inter-, intra-, and RTT-fairness properties substantially compared to pure loss-based approaches (Reno, HS-TCP, STCP). However, the performance of YeAH—similar to all delay-based approaches—can degrade when RTT measurements have significant noise.

VII. Opportunities for Future Research

Currently we have as situation where there is no single congestion control approach for TCP that can universally be applied to all network environments. One of the primary causes is a wide variety of network environments and different (and sometimes opposing) network owners' views regarding which parameters should be optimized. A number of the congestion control algorithms from Section VI (HS-TCP, S-TCP, Africa, TCP-AR, C-TCP, etc.) address this problem by incorporating at least two sets of rules to control the transmission rate of a flow (i.e., conventional Reno-like rules when the network seems to be congested and scalable rules otherwise).

Some algorithms switch modes based on a currently achieved transmission rate (e.g., "reactive" HS-TCP and S-TCP behave as standard Reno while the congestion window is less then a predefined threshold). In some network environments, especially when the probability of loss is high, such switching rules makes algorithms behave in a non-high-speed, therefore inefficient, mode. Other algorithms (e.g., "proactive" Vegas, C-TCP, Illinois, YeAH, etc.) use patterns in delay measurements for switching purposes. If the delay patterns change because of non-congestion-related factors, for example

because of a re-routing path, these algorithms may suffer from efficiency and fairness degradation. This happens because such algorithms do not have the ability to invalidate various internal parameters during the transmission.

Moreover, the current version of Linux kernel provides an API for software developers to choose any one of the supported algorithms for a particular connection. However, there are not yet the well-defined and broadly-accepted criteria to serve as a good baseline for appropriately selecting a congestion control algorithm. Additionally, objective guidelines to select a proper congestion control for a concrete network environment are yet to be defined.

Another aspect of congestion control not yet fully investigated is the problem of short-lived flows (e.g., DNS requests/responses via TCP). The congestion control techniques developed so far do not really work if the connection lifetime is only one or two RTTs. The only congestion control parameter useful during such connections is the initial value of the congestion window. Clearly this value has a direct impact on the performance of short-lived flows. However, if the initial value is large enough and the number of short-lived flows in the network is substantial, the available capacity can easily be exceeded. Because all TCP flows behave independently, a new short-lived flow has no idea of the present network state and can only exacerbate any congestion present in the system. Thus, we need a mechanism to make new flows aware of the current network state (e.g., an ability to estimate the available network path capacity before the actual data transfer). One potential direction to solving this problem is to maintain global estimates of network states. The challenge is distinguishing the independent network states without knowing the exact topology of the Internet.

There are also questions about the fundamental assumptions of TCP congestion control. First of all, it was initially assumed that each TCP flow should be fair to each other. Often, Jain's index (see Section II-A) is used as a fairness measure. However, Jain's metric was based on user shares [23], but everything in TCP is based on individual flows. Essentially, this allows users to game even the "ideally" fair TCP congestion control system and acquire an advantage in the network resources distribution. For example, if one user opens only one TCP connection and another opens five, the network resources will be distributed as one to five. Thus, "ideally" fair congestion control under current definitions is not really "ideal." To summarize, a fundamental research question is how to enforce fairness on a user-level basis without sacrificing throughput of individual flows.

Several problems have arisen as user mobility significantly increased. More and more users now have multiple physical access channels to the Internet. However, TCP is fundamentally unable to use them simultaneously, for example to speed-up data transfer (since a TCP connection is identified by the tuple {srcIP, srcPort, dstIP, dstPort}). A new generation of the reliable data transfer protocol, SCTP [89], provides basic support for multi-homing, but problems of efficient channel utilization, reliable detection of congestion events in separate and common network paths, and user fairness are still to be solved.

Recently, a new congestion control-related problem has appeared on the Internet. For many years, there has been a belief that the more data packets routers can buffer, the more effectively network channels are utilized. The best known recommendation is to set the buffer size equal to a bandwidth-delay product (BDP) of the connection served [90]. In practice, router manufacturers and network administrators often choose maximum values for the bandwidth and delay (or choose some large buffer size). As a result, instead of providing fast feedback to a TCP sender by dropping a number of packets, routers extensively buffer packets making a TCP sender unaware of an abnormal situation in the network. Recently there were several reports of the *excessive buffering syndrome* (also known as a congestive queueing or buffer madness event) on the "End-to-end" mailing list [91], where round-trip delays grew in excess of 5–10 seconds.

VIII. Conclusion

In this work we have presented a survey of various approaches to TCP congestion control that do not rely on any explicit signaling from the network. The survey highlighted the fact that the research focus has changed with the development of the Internet, from the basic problem of eliminating the congestion collapse phenomenon to problems of using available network resources effectively in different types of environments (wired, wireless, high-speed, long-delay, etc.).

In the first part of this survey, we classified and discussed proposals that build a foundation for host-to-host congestion control principles. The first proposal, Tahoe, introduces the basic technique of gradually probing network resources and relying on packet loss to detect that the network limit has been reached. Unfortunately, although this technique solves the congestion problem, it creates a great deal of inefficient use of the network. As we showed, solutions to the efficiency problem include algorithms that (1) refine the core congestion control principle by making more optimistic assumptions about the network (Reno, NewReno); or (2) refine the TCP protocol to include extended reporting abilities of the receiver (SACK, DSACK), which allows the sender to estimate the network state more precisely (FACK, RR-TCP); or (3) introduce alternative concepts for network state estimation through delay measurements (DUAL, Vegas, Veno).

The second part of the survey is devoted to a group of congestion control proposals that are focused on environments where packets are frequently reordered. These proposals show that in such environments, efficiency can be improved significantly by (1) delaying the control actions (TD-FR), or (2) by undoing previously applied actions if reordering is detected (Eifel, DOOR), or (3) by refining the network state estimation heuristic (PR, RR).

In the third part of our survey, we showed that basic host-to-host congestion control principles can solve not only the direct congestion problem but also provide a simple traffic prioritizing feature. Two algorithms examined (Nice and LP), applying slightly different techniques to achieve the same goal, have the same aim: to provide an opportunity to send non-critical data reliably without interfering with other data transfers.

In the last two sections of the survey, we showed that technology advances have introduced new challenges for TCP congestion control. First, we discussed several solutions (the Westwood-family algorithms) which apply similar techniques for estimating the last "good" flow rate and using this rate as a baseline to distinguish between congestion or random packet loss. Second, we reviewed a group of solutions with the most research interest over the recent past. These proposals aim to solve the problem of poor utilization of high-speed or long-delay network channels by TCP flows. The first proposals addressing this problem (HS-TCP, STCP, H-TCP) introduced simple but highly optimistic (aggressive) policies to probe networks for the available resources. Unfortunately, such techniques led to the appearance of a number of other problems, including the intra-, inter-, and RTT-unfairness.

Later proposals employed more intelligent techniques to make congestion control aggressive only when the network is considered congestion-free and conservative during a congestion state. Two proposals, BIC and CUBIC, use packet loss to establish an approximated network resource limit, which is used as a secondary criterion to estimate the current network state. Another group of proposals (FAST, Africa, TCP-AR, C-TCP, Libra, Illinois, Fusion, YeAH) perform this by relying on secondary delay-based network state estimation techniques. Unfortunately, there are disadvantages to both of these approaches, and there is no current consensus in the research community regarding which approach is superior. Not surprisingly, they co-exist in the current Internet: C-TCP is deployed in the Windows-world, and the Linux-world uses CUBIC.

IX. Acknowledgment

The authors are very much obliged to Erik Kline and Janice Wheeler for the valuable input on the survey organization and sleepless nights spent in reading and correcting errors in this text.

References

[1] J. Postel, "RFC793—transmission control protocol," *RFC*, 1981.

[2] C. Lochert, B. Scheuermann, and M. Mauve, "A survey on congestion control for mobile ad hoc networks," *Wireless Communications and Mobile Computing*, vol. 7, no. 5, p. 655, 2007.

[3] J. Postel, "RFC791—Internet Protocol," *RFC*, 1981.

[4] M. Gerla and L. Kleinrock, "Flow control: a comparative survey," *IEEE Transactions on Communications*, vol. 28, no. 4, pp. 553–574, April 1980.

[5] J. Nagle, "RFC896—Congestion control in IP/TCP internetworks," *RFC*, 1984.

[6] C. A. Kent and J. C. Mogul, "Fragmentation considered harmful," in *Proceedings of the ACM workshop on Frontiers in computer communications technology (SIGCOMM)*, Stowe, Vermont, August 1987, pp. 390–401.

[7] S. Floyd and K. Fall, "Promoting the use of end-to-end congestion control in the Internet," *IEEE/ACM Transactions on networking*, vol. 7, no. 4, pp. 458–472, August 1999.

[8] V. Jacobson, "Congestion avoidance and control," *ACM SIGCOMM*, pp. 314–329, 1988.

[9] Z. Wang and J. Crowcroft, "Eliminating periodic packet losses in 4.3–Tahoe BSD TCP congestion control," *ACM Computer Communication Review*, vol. 22, no. 2, pp. 9–16, 1992.

[10] V. Jacobson, "Modified TCP congestion avoidance algorithm," email to the end2end list, April 1990.

[11] M. Allman, V. Paxson, and W. Stevens, "RFC2581—TCP congestion control," *RFC*, 1999.

[12] S. Floyd and T. Henderson, "RFC2582—the NewReno modification to TCP's fast recovery algorithm," *RFC*, 1999.

[13] S. Floyd, T. Henderson, and A. Gurtov, "RFC3782—the NewReno modification to TCP's fast recovery algorithm," *RFC*, 2004.

[14] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanov, "RFC2018—TCP selective acknowledgment options," *RFC*, 1996.

[15] M. Mathis and J. Mahdavi, "Forward acknowledgement: refining tcp congestion control," in *Proceedings of the conference on applications, technologies, architectures, and protocols for computer communications (SIGCOMM)*, New York, NY, USA, 1996, pp. 281–291.

[16] L. Brakmo and L. Peterson, "TCP Vegas: end to end congestion avoidance on a global Internet," *IEEE Journal on Selected Areas in Communication*, vol. 13, no. 8, pp. 1465–1480, October 1995.

[17] G. Hasegawa, K. Kurata, and M. Murata, "Analysis and improvement of fairness between TCP Reno and Vegas for deployment of TCP Vegas to the Internet," in *Proceedings of IEEE ICNP*, 2000, pp. 177–186.

[18] C. P. Fu and S. C. Liew, "TCP Veno: TCP enhancement for transmission over wireless access networks," *IEEE Journal on Selected Areas in Communication*, vol. 21, no. 2, February 2003.

[19] K. Srijith, L. Jacob, and A. Ananda, "TCP Vegas-A: Improving the performance of TCP Vegas," *Computer Communications*, vol. 28, no. 4, pp. 429–440, 2005.

[20] P. Karn and C. Partridge, "Improving round-trip time estimates in reliable transport protocols," in *Proceedings of SIGCOMM*, 1987.

[21] R. Braden, "RFC1122—Requirements for Internet Hosts - Communication Layers," *RFC*, 1989.

[22] W. Stevens, "RFC2001—TCP Slow Start, Congestion Avoidance, Fast Retransmit," *RFC*, 1997.

[23] D.-M. Chiu and R. Jain, "Analysis of the increase and decrease algorithms for congestion avoidance in computer networks," *Computer Networs and ISDN Systems*, vol. 17, no. 1, pp. 1–14, 1989.

[24] V. Jacobson, R. Braden, and D. Borman, "RFC1323—TCP Extensions for High Performance," *RFC*, 1992.

[25] J. Sing and B. Soh, "TCP New Vegas: Improving the Performance of TCP Vegas Over High Latency Links," in *Proceedings of the 4th IEEE International Symposium on Network Computing and Applications (IEEE NCA05)*, 2005, pp. 73–80.

[26] V. Paxson, "End-to-end Internet packet dynamics," *SIGCOMM Computer Communication Review*, vol. 27, no. 4, pp. 139–152, 1997.

[27] M. Przybylski, B. Belter, and A. Binczewski, "Shall we worry about packet reordering," *Computational Methods in Science and Technology*, vol. 11, no. 2, pp. 141–146, 2005.

[28] K. Nichols, S. Blake, F. Baker, and D. Black, "RFC2574—definition of the differentiated services field (DS field) in the IPv4 and IPv6 headers," *RFC*, 1998.

[29] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss, "RFC2575—an architecture for differentiated services," *RFC*, 1998.

[30] T. Bu and D. Towsley, "Fixed point approximations for TCP behavior in an AQM network," in *Proceedings of SIGMETRICS*, New York, NY, USA, 2001, pp. 216–225.

[31] C. Hollot, V. Misra, D. Towsley, and W.-B. Gong, "On designing improved controllers for AQM routers supporting TCP flows," in *Proceedings of IEEE INFOCOM*, vol. 3, 2001, pp. 1726–1734.

[32] J. Bennett, C. Partridge, and N. Shectman, "Packet reordering is not pathological network behavior," *IEEE/ACM Transactions on Networking*, vol. 7, no. 6, pp. 789–798, December 1999.

[33] C. M. Arthur, A. Lehane, and D. Harle, "Keeping order: Determining the effect of tcp packet reordering," in *Proceedings of the Third International Conference on Networking and Services (ICNS)*, June 2007.

[34] J. Arkko, B. Briscoe, L. Eggert, A. Feldmann, and M. Handley, "Dagstuhl perspectives workshop on end-to-end protocols for the future internet," *SIGCOMM Computer Communication Review*, vol. 39, no. 2, pp. 42–47, 2009.

[35] R. Ludwig and R. H. Katz, "The Eifel algorithm: making TCP robust against spurious retransmissions," *SIGCOMM Computer Communication Review*, vol. 30, no. 1, pp. 30–36, 2000.

[36] R. Ludwig and A. Gurtov, "RFC4015—the Eifel response algorithm for TCP," *RFC*, 2005.

[37] F. Wang and Y. Zhang, "Improving TCP performance over mobile ad-hoc networks with out-of-order detection and response," in *Proceedings of the 3rd ACM international symposium on mobile ad hoc networking & computing*, New York, NY, 2002, pp. 217–225.

[38] S. Bohacek, J. Hespanha, J. Lee, C. Lim, and K. Obraczka, "TCP-PR: TCP for Persistent Packet Reordering," in *Proceedings of the International Conference on Distributed Computing Systems*, vol. 23, 2003, pp. 222–233.

[39] S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky, "RFC2883—An Extension to the Selective Acknowledgement (SACK)," *RFC*, 2000.

[40] M. Zhang, B. Karp, S. Floyd, and L. Peterson, "RR-TCP: a reordering-robust TCP with DSACK," International Computer Science Institute, Tech. Rep. TR-02-006, July 2002.

[41] ——, "RR-TCP: a reordering-robust TCP with DSACK," in *Proceedings of the 11th IEEE International Conference on Network Protocols (ICNP)*, 2003, pp. 95–106.

[42] A. Kuzmanovic and E. Knightly, "TCP-LP: low-priority service via end-point congestion control," *IEEE/ACM Transactions on Networking*, vol. 14, no. 4, pp. 739–752, 2006.

[43] D. Clark and W. Fang, "Explicit allocation of best-effort packet delivery service," *IEEE/ACM Transactions on Networking*, vol. 6, no. 4, pp. 362–373, 1998.

[44] X. Xiao and L. Ni, "Internet QoS: A big picture," *IEEE Network*, vol. 13, no. 2, pp. 8–18, 1999.

[45] B. Davie, "Deployment experience with differentiated services," in *Proceedings of the ACM SIGCOMM workshop on Revisiting IP QoS: What have we learned, why do we care?*, New York, NY, 2003, pp. 131–136.

[46] A. Venkataramani, R. Kokku, and M. Dahlin, "TCP Nice: A Mechanism for Background Transfers," *Operating Systems Review*, vol. 36, pp. 329–344, 2002.

[47] A. Kuzmanovic and E. W. Knightly, "TCP-LP: a distributed algorith for low priority data transfer," in *Proceedings of IEEE INFOCOM*, April 2003.

[48] J.-H. Choi and C. Yoo, "One-way delay estimation and its application," *Computer Communications*, vol. 28, no. 7, pp. 819–828, 2005.

[49] A. Kuzmanovic, E. Knightly, and R. Les Cottrell, "HSTCP-LP: A protocol for low-priority bulk data transfer in high-speed high-RTT networks."

[50] K. Ramakrishnan, S. Floyd, and D. Black, "RFC3168—the addition of explicit congestion notification (ECN)," *RFC*, 2001.

[51] M. Gast and M. Loukides, *802.11 wireless networks: the definitive guide*. O'Reilly & Associates, Inc. Sebastopol, CA, USA, 2002, chapter 2.

[52] H. Balakrishnan, S. Seshan, and R. Katz, "Improving reliable transport and handoff performance in cellular wireless networks," *Wireless Networks*, vol. 1, no. 4, pp. 469–481, 1995.

[53] A. Bakre and B. Badrinath, "I-TCP: Indirect TCP for Mobile Hosts," Department of Computer Science, Rutgers University, Tech. Rep. DCS-TR-314, 1994.

[54] K. Brown and S. Singh, "M-tcp: Tcp for mobile cellular networks," *SIGCOMM Computer Communication Review*, vol. 27, no. 5, pp. 19–43, 1997.

[55] S. Mascolo, C. Casetti, M. Gerla, M. Y. Sanadidi, and R. Wang, "TCP Westwood: Bandwidth estimation for enhanced transport over wireless links," in *Proceedings of ACM MOBICOM*, 2001, pp. 287–297.

[56] M. Gerla, M. Y. Sanadidi, and C. E., "Method and apparatus for TCP with faster recovery," U.S. Patent 7 299 280, November 20, 2007.

[57] L. A. Grieco and S. Mascolo, "Performance evaluation and comparison of Westwood+, New Reno and Vegas TCP congestion control," *ACM Computer Communication Review*, vol. 342, April 2004.

[58] R. Wang, M. Valla, M. Sanadidi, B. Ng, and M. Gerla, "Efficiency/friendliness tradeoffs in TCP Westwood," *Proceedings of the Seventh International Symposium on Computers and Communications*, pp. 304–311, 2002.

[59] R. Wang, M. Valla, M. Sanadidi, and M. Gerla, "Adaptive bandwidth share estimation in TCP Westwood," in *Proceedings of IEEE GLOBECOM*, vol. 3, November 2002, pp. 2604–2608.

[60] G. Yang, R. Wang, M. Sanadidi, and M. Gerla, "TCPW with bulk repeat in next generation wireless networks," *IEEE International Conference on Communications 2003*, vol. 1, pp. 674–678, May 2003.

[61] H. Shimonishi, M. Sanadidi, and M. Gerla, "Improving efficiency-friendliness tradeoffs of TCP in wired-wireless combined networks," in *Proceedings of IEEE ICC*, vol. 5, May 2005, pp. 3548–3552.

[62] L. Zhang, S. Shenker, and D. Clark, "Observations on the dynamics of a congestion control algorithm: The effects of two-way traffic," *ACM SIGCOMM Computer Communication Review*, vol. 21, no. 4, pp. 133–147, 1991.

[63] N. Samaraweera, "Non-congestion packet loss detection for TCP error recovery using wireless links," *IEEE Proceedings of Communications*, vol. 146, no. 4, pp. 222–230, August 1999.

[64] S. Cen, P. Cosman, and G. Voelker, "End-to-end differentiation of congestion and wireless losses," *IEEE/ACM Transactions on Networking*, vol. 11, no. 5, pp. 703–717, 2003.

[65] S. Floyd, "RFC3649—HighSpeed TCP for large congestion windows," *RFC*, 2003.

[66] T. Kelly, "Scalable TCP: improving performance in highspeed wide area networks," *Computer Communications Review*, vol. 32, no. 2, April 2003.

[67] S. Floyd, *HighSpeed TCP and Quick-Start for Fast Long-Distance HighSpeed TCP and Quick-Start for fast longdistance networks (slides)*, TSVWG, IETF, March 2003.

[68] D. Leith and R. Shorten, "H-TCP: TCP for high-speed and long-distance networks," in *Proceedings of PFLDnet*, 2004.

[69] D. Leith, "H-TCP: TCP congestion control for high bandwidth-delay product paths," IETF Internet Draft, http://tools.ietf.org/html/draft-leith-tcp-htcp-06, 2008.

[70] C. Caini and R. Firrincieli, "TCP Hybla: a TCP enhancement for heterogeneous networks," *International Journal of Satellite Communications and Networking*, vol. 22, pp. 547–566, 2004.

[71] L. Xu, K. Harfoush, and I. Rhee, "Binary increase congestion control for fast, long distance networks," in *Proceedings of IEEE INFOCOM*, vol. 4, March 2004, pp. 2514–2524.

[72] I. Rhee and L. Xu, "CUBIC: a new TCP-friendly high-speed TCP variant," *SIGOPS Operating Systems Review*, vol. 42, no. 5, pp. 64–74, July 2008.

[73] C. Jin, D. Wei, S. Low, G. Buhrmaster, J. Bunn, D. Choe, R. Cottrel, J. Doyle, W. Feng, O. Martin, H. Newman, F. Paganini, S. Ravot, and S. Singh, "FAST TCP: from theory to experiments," December 2003.

[74] C. Jin, D. Wei, S. Low, J. Bunn, H. Choe, J. Doyle, H. Newman, S. Ravot, S. Singh, F. Paganini *et al.*, "FAST TCP: From Theory to Experiments," *IEEE Network*, vol. 19, no. 1, pp. 4–11, 2005.

[75] D. X. Wei, C. Jin, S. H. Low, and S. Hegde, "FAST TCP: motivation, architecture, algorithms, performance," *IEEE/ACM Transactions on Networking*, vol. 14, no. 6, pp. 1246–1259, 2006.

[76] G. Marfia, C. Palazzi, G. Pau, M. Gerla, M. Sanadidi, and M. Roccetti, "TCP Libra: Exploring RTT-Fairness for TCP," UCLA Computer Science Department, Tech. Rep. UCLA-CSD TR-050037, 2005.

[77] H. Shimonishi and T. Murase, "Improving efficiency-friendliness tradeoffs of TCP congestion control algorithm," in *Proceedings of IEEE GLOBECOM*, 2005.

[78] K. Kaneko, T. Fujikawa, Z. Su, and J. Katto, "TCP-Fusion: a hybrid congestion control algorithm for high-speed networks," in *Proceedings of PFLDnet*, ISI, Marina Del Rey (Los Angeles), California, February 2007.

[79] R. King, R. Baraniuk, and R. Riedi, "TCP-Africa: an adaptive and fair rapid increase rule for scalable TCP," in *Proceedings of IEEE INFOCOM*, vol. 3, March 2005, pp. 1838–1848.

[80] K. Tan, J. Song, Q. Zhang, and M. Sridharan, "A compound TCP approach for high-speed and long distance networks," July 2005.

[81] S. Liu, T. Basar, and R. Srikant, "TCP-Illinois: A loss and delay-based congestion control algorithm for high-speed networks," in *Proceedings of the First International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS)*, 2006.

[82] A. Baiocchi, A. P. Castellani, and F. Vacirca, "YeAH-TCP: yet another highspeed TCP," in *Proceedings of PFLDnet*, ISI, Marina Del Rey (Los Angeles), California, February 2007.

[83] S. Floyd, "RFC3742—Limited slow-start for TCP with large congestion windows," *RFC*, 2004.

[84] A. Aggarwal, S. Savage, and T. Anderson, "Understanding the performance of TCP pacing," in *Proceedings of IEEE INFOCOM*, vol. 3, March 2000, pp. 1157–1165.

[85] S. Ha, Y. Kim, L. Le, I. Rhee, and L. Xu, "A step toward realistic performance evaluation of high-speed TCP variants," in *Fourth International Workshop on Protocols for Fast Long-Distance Networks*, Nara, Japan, March 2006.

[86] S. Belhaj, "VFAST TCP: an improvement of FAST TCP," in *Proc. Tenth International Conference on Computer Modeling and Simulation*, 2008.

[87] R. Kapoor, L.-J. Chen, L. Lao, M. Gerla, and M. Y. Sanadidi, "CapProbe: A simple and accurate capacity estimation technique," in *Proceedings of SIGCOMM*, Portland, Oregon, USA, August/September 2004.

[88] D. Wei, P. Cao, and S. Low, "TCP Pacing Revisited," in *Proceedings of IEEE INFOCOM*, 2006.

[89] A. Caro Jr, J. Iyengar, P. Amer, S. Ladha, and K. Shah, "SCTP: a proposed standard for robust internet data transport," *Computer*, vol. 36, no. 11, pp. 56–63, 2003.

[90] A. Dhamdhere and C. Dovrolis, "Open issues in router buffer sizing," *ACM SIGCOMM Computer Communication Review*, vol. 36, no. 1, pp. 87–92, 2006.

[91] "End-to-end mailing list," http://www.postel.org/e2e.htm.

Alexander Afanasyev received his B.Tech. and M.Tech. degrees in Computer Science from Bauman Moscow State Technical University, Moscow, Russia in 2005 and 2007, respectively. In 2006 he received the medal for the best student scientific project in Russian universities.

He is currently working towards his Ph.D. degree in computer science at the University of California, Los Angeles in the Laboratory for Advanced System Research. His research interests include network systems, network security, mobile systems, multimedia systems, and peer-to-peer environments.

Neil Tilley received his bachelor's degree from the University of California, Davis. His current research interests include parallel and networked systems. He has been pursuing a Ph.D. in Computer Science at the University of California, Los Angels since 2009.

Peter Reiher received his B.S. in Electrical Engineering and Computer Science from the University of Notre Dame in 1979. He received his M.S. and Ph.D. in Computer Science from UCLA in 1984 and 1987, respectively. He has done research in the fields of distributed operating systems, network and distributed systems security, file systems, ubiquitous computing, mobile computing, and optimistic parallel discrete event simulation. Dr. Reiher is an Adjunct Professor in the Computer Science Department at UCLA.

Leonard Kleinrock received his B.E.E. degree from City College of New York (CCNY) in 1957 and received his Ph.D. from Massachusetts Institute of Technology in 1963. He is a Distinguished Professor of Computer Science at UCLA and served as chairman of the department from 1991 to 1995. He received honorary doctorates from CCNY (1997), the University of Massachusetts, Amherst (2000), the University of Bologna (2005), Politecnico di Torino (2005), and the University of Judaism (2007). He has published more than 250 papers and authored six books on a wide array of subjects including queuing theory, packet switching networks, packet radio networks, local area networks, broadband networks, gigabit networks, nomadic computing, peer-to-peer networks and intelligent agents. He is a member of the American Academy of Arts and Sciences, the National Academy of Engineering, an IEEE Fellow, an ACM Fellow, and a founding member of the Computer Science and Telecommunications Board of the National Research Council. Among his many honors, he is the recipient of the CCNY Townsend Harris Medal, the CCNY Electrical Engineering Award, the Marconi Award, the L.M. Ericsson Prize, the NAE Charles Stark Draper Prize, the Okawa Prize, the Communications and Computer Prize, NEC C&C, the IEEE Internet Millennium Award, the UCLA Outstanding Teacher Award, the Lanchester Prize, the ACM SIGCOMM Award, the Sigma Xi Monie Ferst Award, the INFORMS Presidents Award, and the IEEE Harry Goode Award. He was listed by the Los Angeles Times in 1999 as among the "50 People Who Most Influenced Business This Century." He was also listed as among the 33 most influential living Americans in the December 2006 Atlantic Monthly. Kleinrock's work was further recognized when he received the 2007 National Medal of Science, the highest honor for achievement in science bestowed by the President of the United States. This Medal was awarded "for fundamental contributions to the mathematical theory of modern data networks, for the functional specification of packet switching which is the foundation of the Internet Technology, for mentoring generations of students and for leading the commercialization of technologies that have transformed the world."

Appendix A. Evolutionary graph of variants of TCP congestion control