Purdue University
Purdue e-Pubs

Computer Science Technical Reports

Department of Computer Science

2001

TCP Congestion Control: Overview and Survey Of Ongoing Research

Sonia Fahmy
Purdue University, fahmy@cs.purdue.edu

Tapan Prem Karwa

Report Number:
01-016

Fahmy, Sonia and Karwa, Tapan Prem, "TCP Congestion Control: Overview and Survey Of Ongoing Research" (2001). *Computer Science Technical Reports.* Paper 1513.
http://docs.lib.purdue.edu/cstech/1513

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.

TCP CONGESTION CONTROL: OVERVIEW AND SURVEY OF ONGOING RESEARCH

Sonia Fahmy
Tapan Prem Karwa

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

CSD TR #01-016
September 2001

TCP Congestion Control: Overview and Survey of Ongoing Research

Sonia Fahmy and Tapan Prem Karwa
Department of Computer Sciences
1398 Computer Science Building
Purdue University
West Lafayette, IN 47907-1398
E-mail: {fahmy,tpk}@cs.purdue.edu

Abstract

This paper studies the dynamics and performance of the various TCP variants including TCP Tahoe, Reno, NewReno, SACK, FACK, and Vegas. The paper also summarizes recent work at the IETF on TCP implementation, and TCP adaptations to different link characteristics, such as TCP over satellites and over wireless links.

1 Introduction

The Transmission Control Protocol (TCP) is a reliable connection-oriented stream protocol in the Internet Protocol suite. A TCP connection is like a virtual circuit between two computers, conceptually very much like a telephone connection. To maintain this virtual circuit, TCP at each end needs to store information on the current status of the connection, e.g., the last byte sent. TCP is called connection-oriented because it starts with a 3-way handshake, and because it maintains this state information for each connection. TCP is called a stream protocol because it works with units of bytes.

TCP guarantees that it will deliver data supplied by the application to the other end although the underlying layers (IP) offer only unreliable data delivery. That is, even if the data gets lost, corrupted or duplicated while in transit from one end to the other, or is delivered out of order by the communication system, TCP is responsible for delivering error-free in-order data to the application at the other end. Hence, it is reliable.

This reliability is achieved by assigning a sequence number to each byte of data that is transmitted. It is also required of the receiving TCP to send positive acknowledgments (ACKs) back to the data sender. This acknowledgment mentions the next byte of data expected by the receiver. Data is divided into segments (or packets) not exceeding a maximum segment size (MSS) with default value 536 bytes.

Before the data is delivered to the receiver application, the sequence numbers are used to order the data segments, some of which might have arrived out of order at the destination (IP routes may change). The sequence numbers also help in eliminating duplicate segments. Corruption of data is detected by a checksum which is part of each segment transmitted. The receiver handles corrupted data segments by discarding them. If the segment is corrupted or lost, the sender eventually times out and retransmits the corrupted/lost segment.

In this paper, we give an overview of the various TCP congestion control algorithms, compare their performance, and discuss the recent and ongoing work on these algorithms. We also discuss how these algorithms can be tuned for links with long delays (e.g., satellite networks), asymmetric, lossy and low-speed links (e.g., wireless networks).

2 TCP Congestion Mechanisms

Early TCP implementations followed a simple go-back-n model without any fancy congestion control techniques. In the go-back-n model, up to a window worth of data is transmitted continuously without waiting for an ACK. If the segments are received in order and error-free, the receiver ACKs them. If the sender times out waiting for an ACK, it resends all the segments, starting from the oldest segment which has not be acknowledged.

If data arrives at a router faster than its output link rate, the router buffers it resulting in increased queuing delay. If the router runs out of buffer space, it drops incoming segments. If the sender responds to increased queuing delay by timing out and retransmitting the data, it only increases the network traffic and adds to the congested state of the network. The data throughput becomes so low that the network does very little real work. This condition is called congestion collapse.

The receiving TCP uses ACKs to indicate to the sender the successful receipt of bytes up to a certain sequence number. Only when the sender receives ACKs does it continue sending more data. This is called a "self-clocking mechanism." The self-clocking mechanism is fundamental to the idea of "conservation of packets." A connection is said to be in equilibrium if a full window of data has been transmitted. A connection is conservative if a new segment is put onto the network only after an old one leaves it. Since the receipt of an ACK indicates that a segment has left the network, the ACK acts as a signal to the sender to send more data. As more ACKs arrive, more data is sent which results in more ACKs being sent and so on.

Congestion collapse was first experienced in the Internet in 1986 [10]. An investigation resulted into the design of new congestion control algorithms, now an essential part of TCP.

2.1 Slow Start

In modern TCP implementations, every TCP connection starts off in the "slow start" phase [10]. The slow start algorithm uses a new variable called congestion window (cwnd). The sender can only send the minimum of cwnd and the receiver advertised window which we call rwnd (for receiver flow control). Slow start tries to reach equilibrium by opening up the window very quickly. The sender initially sets cwnd to 1 (RFC 2581 [3] suggests an initial window size value of 2 and RFC 2414 suggests min(4×MSS,max(2×MSS,4380 bytes))), and sending one segment. For each ACK that it receives, the cwnd is increased by one segment. The sender always tries to keep a windows worth of data in transit. Increasing cwnd by one for every ACK results in exponential increase of cwnd over round trips. In this sense, the name slow start is a misnomer.

2.2 Congestion Avoidance

TCP uses another variable ssthresh, the slow start threshold, to ensure cwnd does not increase exponentially forever. Conceptually, ssthresh indicates the "right" window size depending on current network load. The slow start phase continues as long as cwnd is less than ssthresh. As soon as it crosses ssthresh, TCP goes into "congestion avoidance." In congestion avoidance, for each ACK received, cwnd is increased by 1/cwnd segments. This is *approximately* equivalent to increasing the cwnd by one segment in one round trip (an additive increase), if every segment (or every other segment) destination is acknowledged.

The TCP sender assumes congestion in the network when it times out waiting for an ACK. ssthresh is set to max(2, min(cwnd/2, rwnd)) segments, cwnd is set to one, and the system goes to slow start [3]. The halving of ssthresh is a multiplicative decrease. The Additive Increase Multiplicative Decrease (AIMD) system has been shown to be stable. Note that increasing the window only stops when a loss is detected.

2.3 ACK Generation

The receiver may not acknowledge every individual segment. The delayed acknowledgment option is recommended in [3]. This algorithm entails that an ACK be generated for at least every other full-sized segment, and definitely within 500 ms of the arrival of the first unacknowledged packet. Out of order segments should be acknowledged immediately, and no more than one ACK should be generated for every incoming segment.

2.4 Retransmission Timeout

TCP maintains an estimate of the round trip time (RTT), i.e., the time it takes for the segment to travel from the sender to the receiver plus the time it takes for the ACK (and/or any data) to travel from the receiver to the sender. The variable RTO (Retransmit Time Out) maintains the value of the time to wait for an ACK after sending a segment before timing out and retransmitting the segment. If RTO is too small, TCP will retransmit segments unnecessarily. If the estimate is too large, the actual throughput will be low because even if a segment gets lost, the sender will not retransmit until the timer goes off.

An estimate of the mean (smoothed) round trip time is maintained via a low-pass filter. Assume that SR is this smoothed RTT estimate and M is the most recently measured RTT, using a recently received ACK for a non-retransmitted segment. The mean deviation v (a close estimate of the standard deviation) can be computed as:

$$v \leftarrow \beta \times |SR - M| + (1 - \beta) \times v$$

β is set to 1/4 in this equation.

The smoothed RTT is then updated as:

$$SR \leftarrow \alpha \times M + (1 - \alpha) \times SR$$

An α value of 1/8 is suggested.

Jacobson [10] suggests that the RTO be computed as:

$$RTO \leftarrow SR + 4 \times v$$

Note that the TCP retransmission timer has *coarse granularity*. A typical value for that timer granularity is 500 ms (100 ms is another common value). This means (in most implementations) that 4 times the deviation v (the second term in the RTO equation) is rounded up to the nearest 500 ms increment. This has an adverse effect on short RTT connections. For example, a 10 ms RTT connection still waits more than 500 ms for the timeout to trigger. This significantly impacts throughput. A number of research studies with variable RTT connections have been conducted with different timer granularity values to determine the best value.

Karn's timer backoff algorithm is also used. This algorithm doubles the RTO value (backs it off) whenever the retransmission timer expires for a retransmitted packet.

2.5 Duplicate Acknowledgments and Fast Retransmit

As previously mentioned, if a TCP receiver receives an out of order segment, it immediately sends back a duplicate ACK (dupack) to the sender. The duplicate ACK indicates the byte number expected (thus it is easy to infer the last in-order byte successfully received). For example, if segments 0 through 5 have been transmitted and segment 2 is lost, the receiver will send a dupack each time it receives an out of order segment, i.e., when it receives segments 3, 4, and 5. Each of these dupacks indicates that the receiver is expecting segment 2 (or expecting byte 1024 assuming segments 0 and 1 are 512 bytes each).

The fast retransmit algorithm uses these dupacks to make retransmission decisions. If the sender receives n dupacks ($n = 3$ was chosen to prevent spurious retransmissions due to out-of-order delivery), it assumes loss and retransmits the lost segment without waiting for the retransmit timer to go off. TCP then reduces ssthresh as previously described (to half cwnd), and resets cwnd to one segment.

TCP Tahoe was one of the first TCP implementations to include congestion control. It included slow start, congestion avoidance and fast retransmit.

2.6 Fast Recovery and TCP Reno

TCP Reno retained all the enhancements in TCP Tahoe, but incorporated a new algorithm, the fast recovery algorithm. Fast recovery is based on the fact that a dupack indicates that a segment has left the network. Hence, when the sender receives 3 dupacks, it retransmits the lost segment, updates ssthresh, and reduces cwnd as in fast retransmit (by half). Fast recovery, however, keeps track of the number of dupacks received and tries to estimate the amount of outstanding data in the network. It inflates cwnd (by one segment) for each dupack received, thus maintaining the flow of traffic. Thus, fast recovery keeps the self-clocking mechanism alive. The sender comes out of fast recovery when it receives an acknowledgment for the segment whose loss resulted in the duplicate ACKs. TCP then deflates the window by returning it to ssthresh, and enters the congestion avoidance phase.

If multiple segments are lost *in the same window of data*, on most occasions, Reno TCP waits for a retransmission timeout, retransmits the segment and goes into slow start mode. This happens when, for each segment loss, Reno enters fast recovery, reduces its cwnd and aborts fast recovery on the receipt of a partial ACK. (A partial ACK is one which acknowledges some but not all of the outstanding segments.) After multiple such reductions, cwnd becomes so small that there will not be enough dupacks for fast recovery to occur and a timeout will be the only option left.

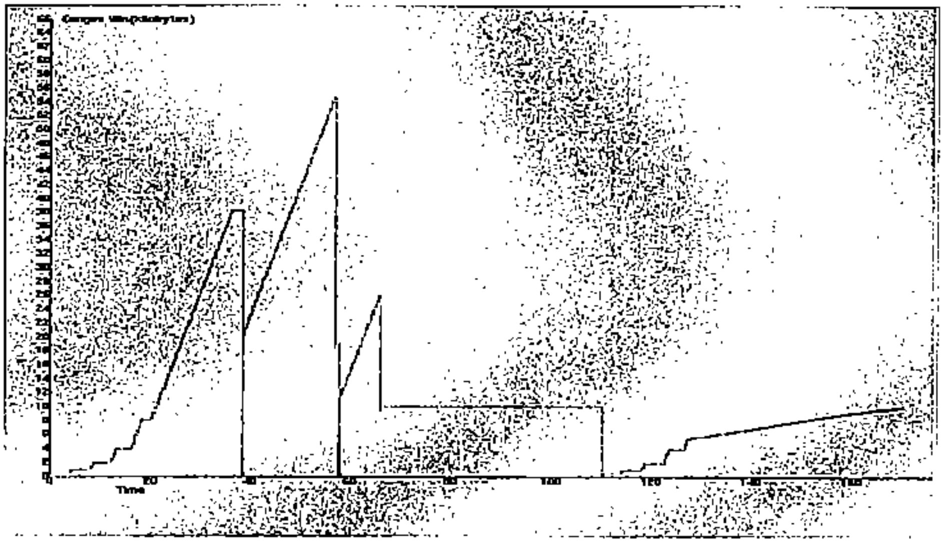

Figure 1: Reno congestion window with multiple consecutive drops

This scenario is illustrated in figures 1 and 2. Figure 1 shows the congestion window changes over time, and figure 2 shows the data sequence number (in red) and ack number (in yellow) over time for a TCP Reno

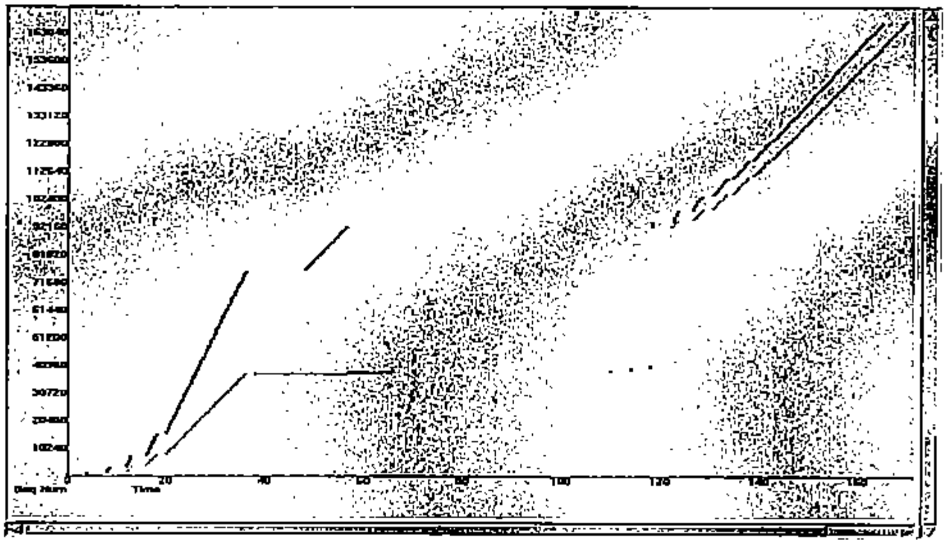

Figure 2: Reno data and ACK sequence numbers with multiple consecutive drops

connection experiencing multiple consecutive losses.

As seen in figure 1, the congestion window grows exponentially (slow start) until some errors occur starting when the congestion window is about 37 kBytes. The sender receives duplicate acknowledgments, and fast retransmit and recovery trigger. ssthresh is set to 18944 bytes. The sender retransmits the lost packet and then sets the congestion window to ssthresh + 3 segments. On receiving subsequent duplicate acks, the congestion window is increased by 512 (the segment size) for every dupack (exponential rise). When the first non-duplicate ACK is received, the congestion window is set to ssthresh. Another set of duplicate acks is received for a subsequent erroneous packet, and fast retransmit and recovery trigger again. ssthresh becomes 9728 bytes. At a certain point, the receiver does not receive any more packets to send dupacks for, and the segment it is expecting is in error, so the sender times out (not enough dupacks for this segment to trigger fast retransmit and recovery). ssthresh is set to half the window size, which is a small number. The sender enters the slow start phase again, starting from a window of one segment. When the window reaches ssthresh, the sender enters the congestion avoidance phase.

Figure 2 show that the sequence numbers increase till the losses occur. When fast retransmit and recovery trigger (after the receipt of 3 dupacks), the first lost segment is retransmitted (red dot in yellow line). More dupacks are received (horizontal yellow line). Later when the window becomes large enough, the sender starts transmitting new segments (the out of order segments were cached). Fast retransmit and recovery trigger again, as seen by the dupacks. Another retransmission occurs, but the sender times out as seen by the long time when there is no transmission. Then it sends only one packet, going into slow start.

2.7 TCP NewReno

A timeout affects the throughput of a connection in two ways. First, the connection has to wait for a timeout to occur and cannot send data during that period of time. Second, after the retransmission timeout occurs, cwnd goes back to one segment. These events adversely affect the performance of the connection

In Reno, partial ACKs bring the sender out of fast recovery resulting in a timeout in case of multiple segment losses. In NewReno TCP, when a sender receives a partial ACK, it does not come out of fast recovery [9, 6, 7]. Instead, it assumes that the segment immediately after the most recently acknowledged segment has been lost, and hence the lost segment is retransmitted. Thus, in a multiple segment loss scenario, NewReno TCP does not wait for a retransmission timeout and continues to retransmit lost segments every time it receives a partial ACK. Thus fast recovery in NewReno begins when three duplicate ACKs are received and ends when either a retransmission timeout occurs or an ACK arrives that acknowledges all of the data up to and including the data that was outstanding when the fast recovery procedure began. Partial ACKs deflate the congestion window by the amount of new data acknowledged, and then add one segment and re-enter fast recovery.

Hoe [9] also suggests two additional algorithms. The first estimates the initial ssthresh by using the delay bandwidth product of the TCP connection (which estimates the number of segments that can be in flight). The second sends a new packet for every 2 duplicate ACKs received during fast recovery. These algorithms are still under investigation and are not part of NewReno as described in RFC 2582 [7].

2.8 TCP with Selective Acknowledgments (SACK)

Another way to deal with multiple segment losses is to tell the sender which segments have arrived at the receiver. Selective Acknowledgments (SACK) TCP does exactly this. The receiver uses each TCP SACK block to indicate to the sender one contiguous block of data that has been received out of order at the receiver. When a SACK blocks are received by the sender, they are used to maintain an image of the receiver queue, i.e., which segments are missing and which have made it to the receiver. Using this information, the sender retransmits only those segments which are missing, without waiting for a retransmission timeout. Only when no segment needs to be retransmitted, new data segments are sent out [6, 12].

The SACK TCP implementation can still use the same congestion control algorithms as Reno (or NewReno). It resorts to the retransmission timeout mechanism to deliver a missing segment to the receiver if ACKs are still not received in time. The main difference between SACK and Reno is the behavior in the event of multiple segment losses. In SACK, just like Reno, when the sender receives 3 dupacks, it goes into fast recovery. The sender retransmits the segment and halves cwnd. SACK maintains a variable called pipe to indicate the number of outstanding segments which are in transit. In SACK, during fast recovery, the sender sends data, new or retransmitted, only when the value of pipe is less than cwnd, i.e., the number of segments in transit are less than the congestion window value. The value of pipe is incremented by one when the sender sends a segment (new or retransmitted) and is decremented by one when the sender receives a duplicate ACK with SACK showing new data has been received. The sender decrements pipe by 2 for partial ACKs [6]. As with NewReno, fast recovery is terminated when an ACK arrives that acknowledges *all* of the data up to and including the data that was outstanding when the fast recovery procedure began.

2.9 Forward Acknowledgments (FACK)

Forward Acknowledgments (FACK) also aims at better recovery from multiple losses. The name "forward ACKs" comes from the fact that the algorithm keeps track of the correctly received data with the highest sequence number. In FACK, TCP maintains 2 additional variables: (1) fack, that represents the forward-most segment that has been acknowledged by the receiver through the SACK option, and (2) retran_data, that reflects the amount of outstanding retransmitted data in the network. Using these 2 variables, the amount of outstanding data during recovery can be estimated as forward-most data sent − forward-most data ACKed (fack value) + outstanding retransmitted data (retran_data value). TCP FACK regulates this value (the amount of outstanding data in the network) to be within one segment of cwnd. cwnd remains constant during the fast recovery period. The fack variable is also used to trigger fast retransmit more promptly [11].

2.10 TCP Vegas

TCP Vegas [5] was presented in 1994 before NewReno, SACK and FACK were developed. Vegas is fundamentally different from other TCP variants in that it does not wait for loss to trigger congestion window reductions. In Vegas, the expected throughput of a connection is estimated to be the number of segments in the pipe, i.e., the number of bytes traveling from the sender to the receiver. To increase throughput, the congestion window must be increased. If congestion exists in the network, the actual throughput will be less than the expected throughput. Vegas uses this idea to decide if it should increase or decrease the window [5].

Vegas keeps track of the time each segment is sent. When an ACK arrives, it estimates RTT as the difference between the current time and the recorded timestamp for the relevant segment. For each connection, Vegas defines BaseRTT to be the minimum RTT seen so far. It calculates the expected throughput as:

$$Expected = WindowSize/BaseRTT$$

where WindowSize is the size of the current congestion window. It then calculates the actual throughput (every RTT) by measuring the RTT for a particular segment in the window and the bytes transmitted in between.

The difference between expected and actual throughputs is maintained in the variable Diff. If Diff $< \alpha$, a linear increase of cwnd takes place in the next RTT; else if Diff $> \beta$, cwnd is linearly decreased in the next RTT. The factors α and β (usually set to 2 and 4) represent too little and too much data in the network, respectively. This is the Vegas congestion avoidance scheme.

Vegas uses a modified slow start algorithm. The original slow start and congestion avoidance need losses to realize the onset of congestion in the network. The modified slow start tries to find the correct window size without incurring a loss. This is done by exponentially increasing its window every *other* RTT and using the other RTT to calculate Diff, when there is no change in the congestion window. Vegas shifts from slow start to congestion avoidance when the actual throughput is lower (by some value γ) than the expected throughput. This addition of congestion detection to slow start gives a better estimate of the bandwidth available to the connection.

Vegas also has a new retransmission policy. A segment is retransmitted after one duplicate ACK (without waiting for 3 dupacks) if the RTT estimate is greater than the timeout value. This helps in those cases where the sender will never receive 3 dupacks because lots of segments within this window are lost or the window size is too small. The same strategy is applied for a non-duplicate ACK after a retransmission.

2.11 Summary

Table 1 summarizes the different TCP variants. The table shows how slow start, congestion avoidance and fast recovery differ, as well as the ACK format required.

3 Effect of Link Characteristics

In wireless networks, the implicit assumption TCP makes that losses indicate network congestion is no longer valid. Losses in wireless networks can result from bit errors and handoffs. There are two different approaches to improve TCP performance in these cases [4]: (1) hide non-congestion-related losses from the TCP sender, using reliable link layer protocols, split connections (separate wireline and wireless TCP

Table 1: TCP Variants

	Tahoe	Reno	NewReno	SACK	FACK	Vegas
In slow start, cwnd updated with every ACK as	cwnd+1	cwnd+1	cwnd+1	cwnd+1	cwnd+1	increase every *other* RTT
In congestion avoidance, cwnd updated with every ACK as	cwnd + 1/cwnd	cwnd + 1/cwnd	cwnd + 1/cwnd	cwnd + 1/cwnd	cwnd + 1/cwnd	linear increase if $\text{expected}-\text{actual}<\alpha$; linear decrease if $>\beta$
Change from slow start to congestion avoidance when	cwnd = ssthresh	cwnd = ssthresh	same, but ssthresh may be estimated	cwnd = ssthresh	cwnd = ssthresh	$\text{expected}-\text{actual}<\gamma$
Fast recovery	none	terminate with partial or full ACK	continue with partial ACK	continue with partial SACKs and send if pipe<cwnd	send as long as outstanding data<cwnd	retransmit with ACK (duplicate or retransmission) if RTT>timeout
ACK format required	ACK	ACK	ACK	SACK	SACK	ACK

connections), and TCP-aware link layer mechanisms including local retransmissions and forward error correction; or (2) adapt the TCP sender to realize that some losses are not due to congestion, using selective acknowledgment and explicit loss notification. TCP-aware reliable link layer mechanisms, selective acknowledgments and explicit loss notification seem to perform best [4].

Recently, the performance implications of link characteristics (PILC) working group at the IETF has recommended certain changes to help TCP adapt to (1) asymmetry, (2) high error rate, and (3) low speed links.

Asymmetry in network bandwidth can result in variability in the ACK feedback returning to the sender. Several techniques can mitigate this effect, including using header compression, reducing ACK frequency by taking advantage of cumulative ACKs, using TCP congestion control for ACKs, giving scheduling priority to ACKs over reverse channel data in routers, applying backpressure with scheduling. The TCP sender must also handle infrequent ACKs. This can be done by bounding the number of back-to-back segment transmissions. Taking into account cumulative ACKs and not number of ACKs at the sender can also improve performance. This scheme is called byte (versus ACK) counting because the sender increases its congestion window based on the number of bytes covered by each ACK. Also reconstructing the ACK stream at the sender; router ACK filtering (removing redundant ACKs from the router queue); or ACK compaction/expansion (conveying information about discarded ACKs from the compacter to the expander) can be used.

For *high error rate* links, experiments show that approaches such as explicit congestion notification (ECN) [13], fast retransmit and recovery and SACK are especially beneficial. Explicit error notification, delayed duplicate acknowledgments, persistent TCP connections, and byte counting are a few of the open research issues in this area. TCP-aware performance enhancing proxies can also be used.

For *low speed* links, in addition to compressing the TCP header and payload, several changes to the congestion avoidance algorithm are recommended. First, hosts that are directly connected to low-speed links should advertise small receiver window sizes to prevent unproductive probing for non-existent bandwidth. Second, maximum transmission units (MTUs) should be carefully selected to not monopolize network interfaces for human-perceptible amounts of time (e.g., 100-200 ms) and to allow delayed acknowledgments. Third, the receiver advertised window size, rwnd, should be carefully selected. Dynamic allocation of TCP buffers (or buffer auto-tuning) based on the current effective window can be used. Finally, binary encoding of web pages, such as the work currently underway at the Wireless Application Protocol (WAP) forum can be used to make web transmissions more compact. Many of the suggested solutions mentioned in this section are still in the research phase.

3.1 TCP over Satellites

In addition to bandwidth asymmetry, restricted available bandwidth, intermittent connectivity, and high error rate due to noise, Geostationary Earth Orbit (GEO) satellite links are characterized by very high latency. This is because such satellites are usually placed at an altitude of around 36,000 km, resulting in a one-way link delay of around 279 ms, or a round trip delay of approximately 558 ms. This results in a long feedback loop and a large delay bandwidth product. For Low Earth Orbit (LEO) and Medium Earth Orbit (MEO) satellites, the delays are shorter, but inter-satellite links are more common and round trip delays are variable.

Allman et al recommend in RFC 2488 [2] several techniques to mitigate the effect of these problems. These techniques include using path MTU discovery, forward error correction (FEC), TCP slow start, congestion avoidance, fast retransmit, fast recovery, and selective acknowledgments (SACK). The TCP window scaling option must also be used to increase the receiver window (rwnd) which places an upper bound on the TCP window size. The algorithms companion to window scaling, including protection against wrapped sequence space (PAWS) and round trip time measurements (RTTM) are also recommended.

A number of additional mitigations are still being researched, and are summarized in RFC 2760 [1]. These include using larger initial window sizes, transaction TCP (eliminates the TCP 3-way handshake with every connection), using multiple TCP connections for a transmission, pacing TCP segment transmissions, persistent TCP connections, byte counting (versus ACK counting) at the sender, ACK filtering, ACK congestion control, explicit error notification, using delayed ACKs only after slow start, setting the initial ssthresh to the delay bandwidth product as in [9], header compression, SACK, FACK, RED, ECN and TCP-friendliness. The next sections discuss RED, ECN and TCP-friendliness in more detail.

Table 2 summarizes the current research issues with link characteristics. A blank entry means the technique was not mentioned in the relevant RFC/draft. Note that slow start, congestion avoidance, fast retransmit and fast recovery are required and SACK is recommended for all cases.

4 Active Queue Management and Explicit Congestion Notification

Since 1993 when Floyd and Jacobson published their Random Early Detection (RED) scheme [8], a lot of attention and research has been focused on refining such router-based drop mechanisms. RED maintains a long term average of the queue length (buffer occupancy) of a router using a low-pass filter. If this average queue length is below a certain minimum threshold, all packets are admitted into the queue. If the average queue length exceeds a certain maximum threshold, all packets are dropped/marked. This early drop helps TCP detect congestion early and allows the router to absorb transient bursts. When the queue length lies in between the minimum and maximum threshold, the packets are dropped/marked with

Table 2: Mitigations for Special Link Characteristics

Technique	Satellites	Asymmetry	High Error Rate	Low Speed
Path MTU discovery	recommended			recommended small
Forward error correction	recommended			
Transaction TCP	recommended			
Larger initial window	recommended			
Delayed ACK after Slow Start	recommended			
Estimating ssthresh	recommended			
Advertised receiver window	large (window scaling)			small/auto-tuned
Byte counting	recommended	recommended	recommended	
Explicit loss notification	recommended		recommended	
Explicit congestion notification	recommended		recommended	
Multiple connections	recommended			
Pacing segments	recommended			
Header and payload compression	recommended	recommended		recommended
Persistent connections	recommended		recommended	
ACK congestion control	recommended	recommended		
ACK filtering	recommended	recommended		
ACK compaction		recommended		
ACK scheduling priority and backpressure		recommended		

a linearly increasing probability up to a maximum drop probability value, p_{max}. This helps avoid TCP synchronization, where TCP drops occur at the same time and TCPs go to slow start at the same time resulting in network underutilization. The long term average avoids bias against bursty traffic. Finally, the dropping/marking is in proportion to the input rate of the TCP connection, which punishes misbehaving sources (sources not implementing congestion avoidance algorithms.

The Explicit Congestion Notification (ECN) option [13], allows active queue management mechanisms such as RED to probabilistically mark (rather than drop) packets when the average queue length falls within the two thresholds, if both the sender and receiver are ECN-capable (determined at connection setup time). In this case, the receiver echoes back to the sender the fact that some of its packets were marked, so the sender knows that the network is approaching a congested state. The sender can thus reduce its congestion window as if the packet was dropped, but need not reduce it drastically, e.g., set it to one or two segments. The sender should only react once per RTT to congestion indications. The sender also informs the receiver of any congestion window reduction so it can stop echoing ECN. The main advantage of this algorithm is that TCP does not have to wait for a timeout and some packet drops can be avoided. Thus loss is not necessary for stopping the window increase.

5 TCP-friendly Congestion Control

Several researchers have investigated how applications can control their transmission rates such that it approximates the behavior of TCP. This allows applications running on top of UDP (that do not require reliability) to co-exist with TCP connections without starving the TCP connections. Different formulae have been developed that try to compute that "TCP-friendly" application rate. The rate is a function of the connection round trip time and the frequency of packet loss indications perceived by the connection.

6 Overview of Other Ongoing Work

Table 3: Important TCP Congestion Control-Related RFCs

RFC Number	Describes
2018	selective acknowledgments (SACK)
2309	random early detection (RED)
2414-6	increasing initial window size
2481	explicit congestion notification
2488	TCP over satellite enhancements
2525	known TCP implementation problems
2581	slow start, congestion avoidance, fast retransmit, fast recovery, idle periods, ACK generation
2582	NewReno
2760	ongoing TCP satellite research

Table 3 summarizes the RFC describing TCP congestion control and their contents. In addition to this, a lot of research is ongoing and a number of papers and IETF drafts are continuously being presented in this active research area. Here are a few of the research items:

- **Idle periods.** After a long idle period (or application-limited period), the TCP ACK clock is no longer useful and TCP can potentially send a burst of size cwnd even though the network conditions have changed. RFC 2581 [3] and Jacobson [10] suggest resetting cwnd to the initial window size (1, 2, or according to RFC 2414) in these cases. There is currently work in progress on decaying the congestion window cwnd by half every RTO interval instead. The value of ssthresh maintains the previous cwnd value in this case.

- **Number of duplicate ACKs.** As indicated in the TCP Vegas algorithm [5], it may be beneficial to respond to the first or second duplicate acknowledgments. Vegas responded with a retransmission if enough time has elapsed. Other strategies are currently being investigated in the IETF community. One mechanism called "limited transmit" suggests sending a new data segment in response to each of the first two duplicate acknowledgments received at the sender. Another mechanism is to sometimes reduce the number of duplicate ACKs required to trigger fast retransmission.

- **SACK Extensions.** There is some work underway to extend SACK to provide more information on order of packet delivery. This can be effective in improving throughput when packets are reordered or replicated, ACKs are lost or transmission timeouts trigger unnecessarily.

7 Acknowledgments

The authors would like to thank Farnaz Erfan and Minseok Kwon for their help with the simulation experiments.

References

[1] M. Allman, S. Dawkins, D. Glover, J. Griner, D. Tran, T. Henderson, , J. Heidemann, S. Ostermann, K. Scott, J. Semke, and J. Touch. Ongoing TCP research related to satellites. RFC 2760, February 2000.

[2] M. Allman, D. Glover, and L. Sanchez. Enhancing TCP over satellite channels using standard mechanisms. RFC 2488, January 1999.

[3] M. Allman, V. Paxson, and W. Stevens. TCP congestion control. RFC 2581, April 1999. Also see http://tcpsat.lerc.nasa.gov/tcpsat/papers.html.

[4] H. Balakrishnan, V. Padmanabhan, S. Seshan, and R. Katz. A comparison of mechanisms for improving TCP performance over wireless links. *IEEE/ACM Transactions on Networking*, 5(6):756–769, December 1997. http://HTTP.CS.Berkeley.EDU/~hari/papers/ton.ps.

[5] L. Brakmo, S. O'Malley, and L. Peterson. TCP vegas: New techniques for congestion detection and avoidance. In *Proceedings of the ACM SIGCOMM*, pages 24–35, August 1994. http://netweb.usc.edu/yaxu/Vegas/Reference/vegas93.ps.

[6] K. Fall and S. Floyd. Simulation-based comparisons of Tahoe, Reno, and SACK TCP. In *ACM Computer Communication Review*, volume 26, pages 5–21, July 1996. ftp://ftp.ee.lbl.gov/papers/sacks.ps.Z.

[7] S. Floyd and T. Henderson. The NewReno modification to TCP's fast recovery algorithm. RFC 2582, April 1999. Also see http://www.aciri.org/floyd/tcp_small.html.

[8] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, August 1993. ftp://ftp.ee.lbl.gov/papers/early.ps.gz.

[9] J. Hoe. Improving the start-up behavior of a congestion control scheme for TCP. In *Proceedings of the ACM SIGCOMM*, pages 270–280, August 1996. http://www.acm.org/sigcomm/ccr/archive/1996/conf/hoe.ps.

[10] V. Jacobson. Congestion avoidance and control. In *Proceedings of the ACM SIGCOMM*, volume 18, pages 314–329, August 1988. ftp://ftp.ee.lbl.gov/papers/congavoid.ps.Z.

[11] M. Mathis and J. Mahdavi. Forward acknowledgment: Refining TCP congestion control. In *Proceedings of the ACM SIGCOMM*, August 1996. Also see http://www.psc.edu/networking/papers/papers.html.

[12] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP selective acknowledgement options. RFC 2481, October 1996.

[13] K. Ramakrishnan and S. Floyd. A proposal to add explicit congestion notification (ECN) to IP. RFC 2481, January 1999.