# A Report on Some Recent Developments in TCP Congestion Control

Sally Floyd

October 9, 2000
To appear in IEEE Communications Magazine, April 2001


## Abstract

This paper discusses several changes to TCP's congestion control, either proposed or in progress. The changes to TCP include a Limited Transmit mechanism for transmitting new packets on the receipt of one or two duplicate acknowledgements, and a SACK-based mechanism for detecting and responding to unnecessary Fast Retransmits or Retransmit Timeouts. These changes to TCP are designed to avoid unnecessary Retransmit Timeouts, to correct unnecessary Fast Retransmits or Retransmit Timeouts resulting from reordered or delayed packets, and to assist the development of viable mechanisms for Corruption Notification. The changes in the network include Explicit Congestion Notification (ECN), which builds upon the addition of Active Queue Management.


## 1 Introduction

The basis of TCP congestion control lies in Additive Increase Multiplicative Decrease (AIMD), halving the congestion window for every window containing a packet loss, and increasing the congestion window by roughly one segment per RTT otherwise. A second component of TCP congestion control of fundamental importance in highly-congested regimes is the Retransmit Timer, including the exponential backoff of the retransmit timer when a retransmitted packet is itself dropped. A third fundamental component is the Slow-Start mechanism for the initial probing for available bandwidth, instead of initially sending at a high rate that might not be supported by the network. The fourth TCP congestion control mechanism is ACK-clocking, where the arrival of acknowledgements at the sender is used to clock out the transmission of new data.

Within this general congestion control framework of Slow-Start, AIMD, Retransmit Timers, and ACK-clocking, there is a wide range of possible behaviors. These include the response when multiple packets are dropped within a round-trip time; the precise algorithm for setting the retransmit timeout; the response to reordered or delayed packets; the size of the initial congestion window; and so on. Thus, different TCP implementations differ somewhat in their ability to compete for available bandwidth, However, because they all adhere to the same underlying mechanisms, there is no bandwidth starvation between competing TCP connections. That is, while bandwidth is not necessarily shared equally between different TCP implementations, it is unlikely that one conformant TCP implementation will prevent another from receiving a reasonable share of the available bandwidth.

The changes to TCP discussed in this paper all adhere to this underlying framework of Slow-Start, AIMD, Retransmit Timers, and ACK-clocking; that is, none of these changes alter the fundamental underlying dynamics of TCP congestion control. Instead, these proposals would help to avoid unnecessary Retransmit Timeouts, correct unnecessary Fast Retransmits and Retransmit Timeouts resulting from reordered or delayed packets, and reduce unnecessary costs (in delay and unnecessary retransmits) associated with the mechanism of congestion notification. These proposals are in various stages of the processes of research, standardization, and deployment.

Other changes to TCP's congestion control mechanisms in various stages of deployment but not discussed in this paper include larger initial windows, and NewReno TCP for greater robustness with multiple packet losses in the absence of the SACK option. Changes to TCP's congestion control mechanisms largely in the research stages include ACK filtering or ACK congestion control for traffic on the return path, a range of improvements to the Slow-Start procedure, and rate-based pacing. Pointers to the literature for many of these proposals can be found in RFC 2760 [3]. Proposals for greater robustness against misbehaving end-hosts (as in [13]) would give protection against a single end node (e.g., at the web client) attempting to subvert end-to-end congestion control, while not changing the congestion control behavior in the case of conformant end-nodes.

Proposals for Endpoint Congestion Management (ECM) would not change the congestion control mechanisms for a single flow, but would change the number of individual transfers treated as a single stream in terms of end-to-end congestion control. Other proposals for more explicit communication between the transport layer and the link layer below or the application level above (e.g., HTTP), or for performance-enhancing proxies, would modify the context of congestion control, and not its underlying mechanisms.

Several themes are carried throughout this paper. One theme is that proposed changes to TCP's congestion control algorithms tend towards increased robustness across a wide range of environments, rather than fine-tuning for one particular environment or traffic type at the expense of another.

A second theme of this paper is that many independent changes are in progress, and evaluating one change requires taking into account its interactions with other changes in progress. In addition to considering the impact of a particular change in TCP given the current environment, with all else held fixed, it is also useful to consider the potential impact of a proposed change some years down the road, when other changes to TCP and to the network are in place.

A third theme is that there is unavoidable heterogeneity in the congestion control behaviors of deployed TCP implementations, in part due to the uneven progress of proposed changes to TCP from research to standardization to actual deployment. As an example of uneven deployment, the SACK option to TCP in RFC 2018 [11], which allows more robust operation when multiple packets are lost from a single window of data, was standardized (as Proposed Standard) in 1996, but is only now becoming widely deployed. (This deployment is documented by the TBIT tool [10] as well as by many other researchers.)

Section 2 of the paper discusses some changes to TCP at the end-nodes, and Section 3 discusses changes in the network that would affect TCP's congestion control behavior, as follows. Section 2.1 discusses the Limited Transmit proposal for reducing unnecessary retransmit timeouts, Section 2.2 discusses the potential of mechanisms based on D-SACK information to add robustness in the presence of reordered or delayed packets, and Section 2.3 discusses one possible path of development for Corruption Notification. In the section on network changes, Section 3.1 first discusses active queue management mechanisms such as RED for controlling the average queue size and reducing unnecessary packet drops. Section 3.2 then discusses ECN, which, building upon active queue management, allows routers the option of marking rather than dropping packets as indications of congestion to the end-nodes.

# 2 Small Changes in TCP's Congestion Control Mechanisms

This section discusses several small changes to TCP's congestion control mechanisms intended to avoid some of the unnecessary Retransmit Timeouts for small transfers, and to improve performance in environments with reordered, delayed, or corrupted packets. Instead of involving fundamental changes to TCP's congestion control, these changes would bring TCP closer to its "pure" congestion control behavior described earlier of ACK-clocking, Slow-Start for starting up, AIMD for congestion windows larger than one segment, and the exponential backoff of the retransmit timer for environments of heavy congestion.

## 2.1 Avoiding Unnecessary Retransmit Timeouts

Retransmit timeouts are a necessary mechanism of last resort in TCP flow control, used when the TCP sender has no other method for determining that a segment must be retransmitted. In addition, the exponential backoff of retransmit timers is a fundamental component of TCP congestion control, of particular importance when the congestion window is at most one segment. However, when the congestion window is larger than one segment, TCP is able to use the basic AIMD congestion control mechanisms, and in this case it would be preferable to avoid unnecessary Retransmit Timeouts as much as possible.

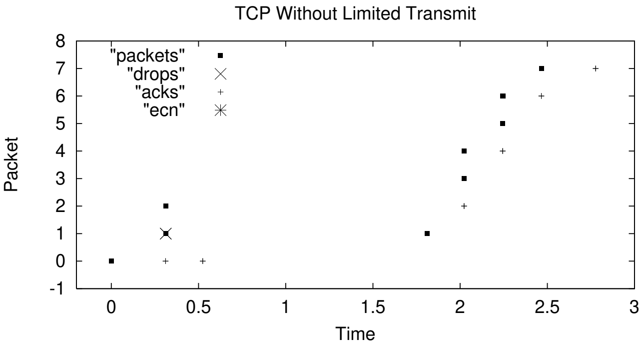


Figure 1: TCP without Limited Transmit, with a single packet drop.

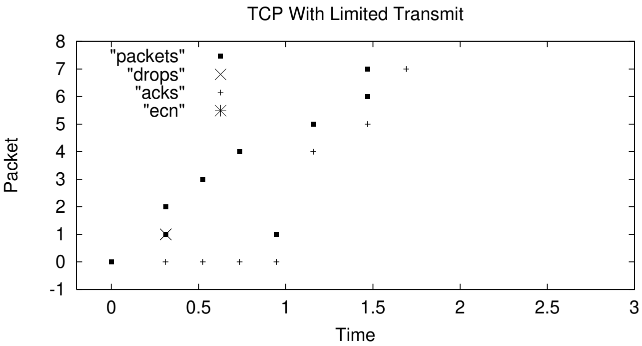


Figure 2: TCP with Limited Transmit, with a single packet drop.

Current TCP implementations have two possible mechanisms for detecting a packet loss, Fast Retransmit or a Retransmit Timeout. A TCP connection generally recovers more promptly from a packet loss with Fast Retransmit, inferring a packet loss after three duplicate ACKs have been received. When Fast Retransmit is invoked, the TCP sender retransmits the segment inferred to be lost, and halves its congestion window, continuing with the data transfer. If the TCP data sender doesn't receive three duplicate ACKs after a loss (for example, because the congestion window was less than four segments), then the TCP sender goes through the possibly-considerable delay of waiting for the transmit timer to expire.

Experimental studies such as those in [4] show that the performance costs to small flows of unnecessarily waiting for a retransmit timer to expire can be considerable. Figure 1 shows a short TCP connection with the second packet dropped in the network. The graph has a mark for each packet transmitted, with time on the $x$-axis and the packet number on the $y$-axis. As shown in the graph, the TCP sender has to wait for a Retransmit Timeout to recover from the packet loss.[1]

A number of researchers have proposed a Limited Transmit mechanism where the sender transmits a new segment after receiving one or two duplicate ACKs, if allowed by the receiver's advertised window; several of the proposals have been described in RFC 2760 [3]. Because the first or second duplicate ACK is evidence that a packet has been delivered to the receiver, and the data sender has not yet determined that a packet has been lost, it is conformant with the spirit of the congestion window to allow a new packet to enter the pipeline. Because the Limited Transmit mechanism transmits a new packet on receiving the first or second duplicate ACK, rather than retransmitting an old packet suspected to have been dropped, the Limited Transmit mechanism is robust to reordered packets.

In many cases the Limited Transmit mechanism allows TCP connections with small windows to recover from less that a full window of packet losses without a Retransmit Timeout. As an example, Figure 2 shows a simulation with Limited Transmit, with the second packet dropped in the network. In this case, when the TCP sender receives a duplicate ACK acknowledging the receipt of the third packet, the sender is able to send a new packet, ultimately resulting in three duplicate ACKs followed by a Fast Retransmit. We would note that in both the simulations with and without Limited Transmit, the TCP sender halves the congestion window in response to the packet drop. However, with Limited Transmit the TCP sender does not have to wait for a Retransmit Timeout to learn of the lost packet. As discussed in Section 3.2, the use of Explicit Congestion Notification (ECN) would also avoid the unnecessary Retransmit Timeout in this case.

The Limited Transmit mechanism has been submitted to the IETF and approved by the Working Group [2], and we hope that it will soon become an accepted part of the TCP specification. This should help in reducing unnecessary retransmit timeouts, while preserving the fundamental role of retransmit timers in congestion control for regimes where the available bandwidth is at most one packet per round-trip time.

[1] These simulations can be run in the NS simulator with the command "./test-all-LimTransmit" in "tcl/test".

## 2.2 'Undoing' Unnecessary Congestion Control Responses to Reordered or Delayed Packets

There are a number of scenarios where a TCP sender can infer a packet loss, and consequently reduce its congestion window, when in fact there has been no loss. When the retransmit timer expires unnecessarily early (that is, when no data or ACK packet has been lost, and the sender would have received acknowledgements for the outstanding packets if it had waited a little longer), then the TCP sender unnecessarily retransmits a segment. More importantly, an early Retransmit Timeout results in an unnecessary reduction of the congestion window, as the flow has not experienced any packet losses. Similarly, when Fast Retransmit is invoked unnecessarily, after three duplicate ACKs have been received due to reordering rather than packet loss, the TCP sender also unnecessarily retransmits a packet and reduces its congestion window.

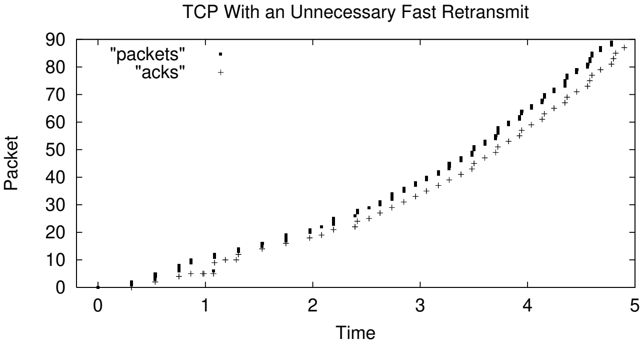


Figure 3: TCP with delayed packets at time 0.75, and an unnecessary Fast Retransmit.

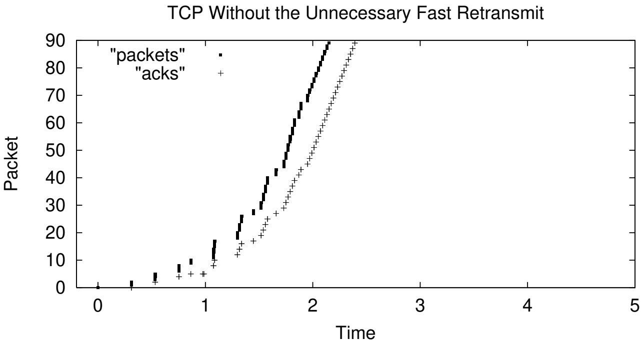


Figure 4: TCP with delayed packets at time 0.75, but without the unnecessary Fast Retransmit.

Figure 3 shows a TCP connection with several packets delayed at time 0.75, so that the TCP connection undergoes an unnecessary Fast Retransmit at time 1.1, accompanied by the termination of Slow-Start. Figure 4 shows a similar simulation with the packets delayed slightly less long, to avoid the unnecessary Fast Retransmit. The second simulation is simply to emphasize the performance damage done by the unnecessary Fast Retransmit in the first simulation.

While it would be possible to fine-tune TCP's Retransmit Timeout algorithms to achieve an improved balance between unnecessary Retransmit Timeouts and unnecessary delay in detecting loss, it is not possible to design Retransmit Timeout algorithms that never result in an unnecessary Retransmit Timeout. Similarly, while it would be possible to fine-tune TCP's Fast Retransmit algorithm to achieve an improved balance between unnecessary Fast Retransmits and unnecessary delay in detecting loss, it is not possible to devise a Fast Retransmit algorithm that always correctly determines, after the receipt of a duplicate ACK, whether or not a packet loss has occurred. Thus, it would be desirable for TCP congestion control to perform well even in the presence of unnecessary Retransmit Timeouts and Fast Retransmits.

For a flow with a large congestion window $W$, an unnecessary halving of the congestion window can be a significant performance penalty, as it takes at least $W/2$ round-trip times for the flow to recover its old congestion window. Similarly, for an environment with persistent reordering of packets within a flow, or for an environment with an unreliable estimated upper bound on the round-trip time, this repeated unnecessary halving of the congestion window can have a significant performance penalty. A persistent reordering of packets in a flow could result from changing routes or from the link-level retransmission of corrupted packets over a wireless link.

An initial step towards adding robustness in the presence of unnecessary Retransmit Timeouts and Fast Retransmits is to give the TCP sender the information to determine when an unnecessary Retransmit Timeout or Fast Retransmit has occurred. This first step has been accomplished with the D-SACK (for duplicate-SACK) extension (RFC 2883 [9]) that has recently been added to the SACK TCP option. The D-SACK extension allows the TCP data receiver to use the SACK option to report the receipt of duplicate segments. With the use of D-SACK, the TCP sender can correctly infer the segments that have been received by the data receiver, including duplicate segments.

When the sender has retransmitted a packet, D-SACK does not allow TCP to distinguish between the receipt at the receiver of both the original and retransmitted packet, and the receipt of two copies of the retransmitted packet, one of which was duplicated in the network. If necessary, TCP's timestamp option could be used to distinguish between these two cases. However, in an environment with minimal packet replication in the network, D-SACK allows the TCP sender to make reasonable inferences, one round-trip time after a packet has been retransmitted, about whether the retransmission was necessary or unnecessary.

If the TCP data sender determines, a round-trip time after retransmitting a packet, that the receiver received two copies of that segment and therefore that the packet retransmission was most likely unnecessary, then one possibility would be for the sender to "undo" the halving in the congestion window. The sender could "undo" a recent halving by setting the Slow-Start threshold ssthresh to the previous value of the old congestion window, effectively re-entering Slow-Start until the congestion window has reached its old value. If the connection had been in Slow-Start when the unnecessary Fast Retransmit was triggered, then ssthresh could be reset to its old value, restoring Slow-Start. In addition to restoring the congestion window, the TCP sender would adjust the duplicate acknowledgement threshold or the retransmit timeout parameters to avoid the wasted bandwidth of persistent unnecessary retransmits.

The first part of this work, providing the information to the sender about duplicate packets received at the receiver, is done with the D-SACK extension. The next step is to evaluate specific mechanisms for identifying an unnecessary halving of the congestion window, and for adjusting the duplicate acknowledgement threshold or retransmit timeout parameters. Once this is done, there is no fundamental reason why TCP congestion control cannot perform effectively in an environment with persistent reordering.

## 2.3 Implications for Corruption Notification

One of the fundamental components of TCP congestion control is that packet losses are used as indications of congestion. TCP halves its congestion window after any window of data in which one or more packets have been lost. With the addition of ECN to the IP architecture, routers would be able to set a bit in the ECN field of the IP header as an indication of congestion, and end nodes would be able to use ECN indications as a second method for indicating congestion. However, the addition of ECN to the IP architecture would not eliminate congestion-related packet losses due to buffer overflow, and therefore would not allow end nodes to ignore packet losses as indications of congestion.

For wired links, packet losses due to packet corruption instead of congestion are infrequent; this is not always the case for wireless links [6]. While many wireless links use Forward Error Correction (FEC) and link-level retransmission to repair packet corruption, it is not always possible to eliminate all packet corruption in a timely fashion.

One possible response to packet corruption would be for the TCP sender to "undo" the congestion window reduction if the TCP sender found out, after the fact, that a single packet loss had been due to corruption rather than congestion. This late "undoing" of a congestion window reduction could use a delayed notification of packet corruption, where the TCP sender receives the notification of corruption some time after it has already retransmitted the packet and halved the congestion window.

Such a mechanism for the late "undoing" of a congestion window reduction would allow a link-level protocol to develop a method for the delayed sending of a corruption notification message to the TCP data receiver. That is, the link-level protocol could determine when the link level is no longer attempting to retransmit a packet has been lost at the link level due to corruption. In this case, the link-level protocol could arrange that the link-level sender send a corruption notification message to the IP destination of the corrupted packet. Of course, this short corruption notification message could itself be corrupted or lost, in which case the transport end nodes would be left to their earlier inference that the packet had been lost due to congestion.

With this form of Congestion Notification, a TCP sender that has halved its congestion window as a result of a single packet loss could receive information from the link level, some time later, that this packet was lost due to corruption rather than due to congestion. If such mechanisms for corruption notification are developed, a necessary next step will be to determine the appropriate response of the end nodes to this corruption. For packet corruption that is not an indication of congestion from competing traffic, halving the congestion window in response to a single corrupted packet is clearly unnecessarily severe. At the same time, maintaining a persistent high sending rate in the presence of a high packet corruption rate is also clearly unacceptable; each corrupted packet could represent wasted bandwidth on the path to the point of corruption.

The development of corruption notification will also require the development of accompanying mechanisms for protection against misbehaving routers or receivers, so that receivers cannot mislead the sender into treating a congestion-related packet loss as a corruption-related loss.

# 3 Changes in the Network

TCP's congestion control behavior is affected by changes in the network as well as by changes to the TCP implementations at the end hosts. In this section we discuss the impact of ECN on TCP congestion control. Because ECN depends on the deployment of Active Queue Management, we first consider the impact of Active Queue Management by itself on TCP congestion control behavior.

The scheduling mechanisms used in the routers also have a significant impact on TCP's congestion control dynamics. However, in this paper we limit our discussion to the environment of FIFO scheduling typical of the current Internet.

## 3.1 Active Queue Management

It has long been known that Drop-Tail queue management can result in pathological packet-dropping patterns, particularly in simple simulation scenarios with long-lived connections, one-way traffic, and fixed packet sizes; this is discussed in detail in [7]. A more relevant issue for actual networks is that with small-scale statistical multiplexing, Drop-Tail queue management can result in global synchronization among multiple TCP connections, with underutilization of the congested link resulting from several connections halving their congestion window at the same time. This global synchronization is less likely to be a problem with large-scale statistical multiplexing.

However, there is a fundamental tradeoff between high throughput and low delay with any queue management, whether it is Active Queue Management such as RED (Random Early Detection) [8] or simple queue management such as Drop-Tail. Maintaining a low average delay with Drop-Tail queue management means that the queue will have little capacity to accommodate transient bursts, and can result in an unnecessarily-high packet drop rate. At the same time, Drop-Tail queue management is perfectly capable of delivering acceptable performance in many circumstances. For example, experimental studies such as [5] have confirmed that with higher levels of statistical multiplexing and the heterogeneity of session start times, round-trip times, transfer sizes, and packet sizes typical of the current Internet, Drop-Tail queue management is quite capable of delivering both high link utilization and low overall response times for web traffic.

The main motivation for Active Queue Management is to control the average queueing delay while at the same time allowing transient fluctuations in the queue size to reduce unnecessary packet drops. For environments where low per-packet delay and high aggregate throughput are both important performance metrics, active queue management can allow a queue to be tuned for low average per-packet delay while reducing the penalty in unnecessary packet drops that might be necessary with Drop-Tail queue management with the same average queueing delay. However, for environments with the same *worst-case* queueing delay for Drop-Tail as for Active Queue Management, the lower *average* queue size maintained by Active Queue Management can sometimes come at the cost of a higher packet drop rate.

In environments with highly bursty packet arrivals (as would be encouraged by a scenario with ACK compression and ACK losses on the return path), Drop-Tail queue management can result in an unnecessarily large number of packet drops, as compared to Active Queue Management, particularly with similar average queueing delays. Even if there is full link utilization, a higher packet drop rate can have two consequences, wasted bandwidth on congested links before the point of loss, and a higher variance in transfer times for the individual flows.

One might ask if unnecessary packet drops really matter, if full link utilization can be maintained. Unnecessary packet losses result in wasted bandwidth to the point of loss only if there are multiple congested links, where other traffic could have made more effective use of the available bandwidth upstream of the point of congestion. Paths with multiple congested links might seem unlikely, given the lack of congestion reported within many backbone networks. However, even with uncongested backbone networks, a path with a congested link to the home, a congested link at an Internet exchange point, and a congested transoceanic link would still be characterized by multiple congested links.

The second possible consequence of unnecessary packet losses even with full link utilization can be a higher variance in transfer times. For example, small flows with an 'unnecessary' packet drop of the last packet in a transfer will have a long wait for a retransmit timeout, while other active flows might have their total transfer time shortened by one packet transmission time.

We would also note that the bursty packet loss patterns typical of Drop-Tail queue management have had a particularly unfortunate interaction with Reno TCP, but as Reno implementations in the Internet are gradually being replaced by NewReno and Sack TCP, this interaction is becoming less of a problem. Reno TCP has well-known performance problems with multiple packets dropped from a single window of data, and these multiple drops are more likely with Drop-Tail than with Active Queue Management. The gradual replacement of Reno by NewReno and Sack TCP does not mean that Active Queue Management is no longer needed, however; it just means that this particular performance problem of multiple packets dropped from a window of data is now of less pressing concern.

## 3.2 Explicit Congestion Notification

ECN allows routers to set the Congestion Experienced (CE) bit in the IP packet header as an indication of congestion to the end nodes as an alternative to dropping the packet. ECN is specified in RFC 2481 [12], and is currently an Experimental addition to the IP architecture. ECN-capable packets in TCP connections advertise their capability for ECN in the IP header. In terms of congestion control, TCP connections respond to a single ECN mark as they would to a single packet loss. One of the key advantages of ECN will not be for TCP traffic, but instead for traffic such as real-time or interactive traffic, where the cost of an unnecessary packet drop is either the unnecessary delay of retransmitting the packet, or possibly making-do without that packet altogether.

To first order, TCP congestion control dynamics with ECN are similar to those without ECN. The main difference is that the TCP sender does not have to retransmit the marked packet (as it would if the packet had been dropped). For example, ECN would mean shorter transfer times for the small number of short flows that might otherwise have the final packet of a transfer dropped. Experimental studies such as [1] have shown the performance advantages of ECN for TCP short transfers.

One of the advantages of ECN is that, by replacing a packet drop by a packet mark, a TCP connection with a small congestion window can avoid a retransmit timeout. Figure 6 shows a simulation with ECN-Capable TCP, with the third packet marked rather than dropped in the network. The TCP sender receives the congestion notification with the receipt of the ACK packet, and halves its congestion window. Figure 5 shows the same simulation with no packets marked or dropped, to emphasize the halving of the congestion window in the simulation with ECN. Figure 1 showed the same scenario with the third packet dropped rather than marked, resulting in a Retransmit Timeout.

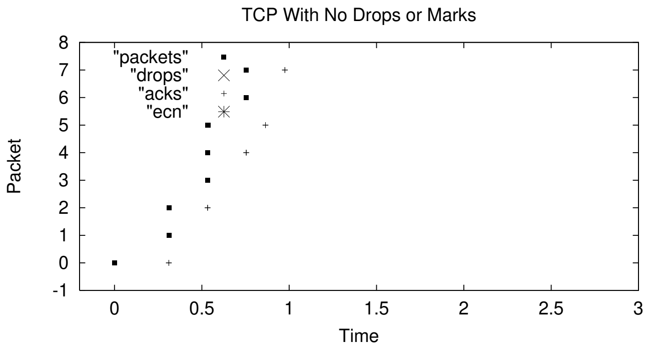


Figure 5: TCP with no packets dropped or marked.

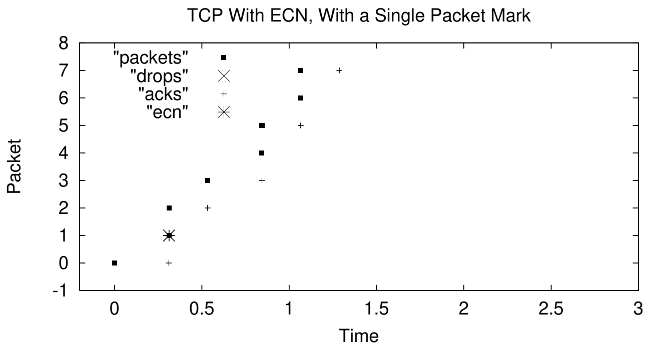


Figure 6: TCP with ECN, with a single packet marked.

Figure 2 shows that Limited Transmit without ECN could also have avoided a Retransmit Timeout in this scenario. Because Limited Transmit could sometimes avoid a Retransmit Timeout in this case even in the absence of ECN, the deployment of Limited Transmit could diminish somewhat the performance benefits of ECN for small flows (by improving TCP performance even in the absence of ECN, not by worsening TCP performance with ECN). Thus, some of the performance advantages reported for ECN for TCP short transfers would diminish with the introduction of Limited Transmit. At the same time, there are many scenarios (e.g., for a transfer of only a few packets) where ECN avoids a Retransmit Timeout while Limited Transmit does not, and also scenarios (with forced drops due to buffer overflow) where the opposite is the case.

Experimental studies such as [1] have also shown that ECN has some performance advantages even for long TCP transfers. One performance advantage is that ECN eliminates the delays of the Fast Retransmit and Retransmit Timeout procedures, allowing the TCP sender to immediately begin transmitting at the reduced rate. ECN gives an explicit notification of congestion that is robust in the presence of reordered or delayed packets, and does not rely on the imprecise duplicate acknowledgement thresholds or retransmit timeout intervals used by TCP to detect lost packets.

As noted earlier, ECN cannot be relied upon to completely eliminate packet losses as indications of congestion, and therefore would not allow the end nodes to interpret packet losses as indications of corruption instead of congestion. Because ECN cannot eliminate packet loss completely, it does not eliminate the need for Limited Transmit. Similarly, ECN does not eliminate the need for Fast Retransmit and Retransmit Timeout mechanisms to detect dropped packets, and therefore does not eliminate the need for the D-SACK procedures discussed in Section 2.2 for undoing unnecessary congestion control responses to reordered or delayed packets.

# 4 Conclusions

To summarize, changes to TCP are in progress that would continue to bring TCP's congestion control behavior closer to the goal of AIMD for larger congestion windows, and exponential backoff of the retransmit timer for regimes of higher congestion. These changes include the Limited Transmit mechanism to avoid unnecessary Retransmit Timeouts, and D-SACK-based mechanisms to identify and reverse unnecessary congestion control responses to reordered or delayed packets. More speculative possibilities include corruption notification messages for the link level to inform transport end-nodes about packets lost to corruption rather than congestion.

At the same time, changes in the network are either proposed or in progress to reduce unnecessary packet losses, and to replace some congestion-related losses by packet marking instead. Like the possible changes to TCP, changes such as ECN would bring TCP's congestion control behavior closer to its desired ideal behavior, as well as being of great potential value to newer unreliable unicast, unreliable multicast, and reliable multicast transport protocols.

# References


[1] U. Ahmed and J. Salim. Performance Evaluation of Explicit Congestion Notification (ECN) in IP Networks. RFC 2884, Jul. 2000.

[2] M. Allman, H. Balakrishnan, and S. Floyd. Enhancing TCP's Loss Recovery Using Early Duplicate Acknowledgment Response. Internet draft draft-allman-tcp-lossrec-00.txt, work-in-progress, Jun. 2000.

[3] M. Allman, S. Dawkins, D. Glover, J. Griner, D. Tran, T. Henderson, J. Heidemann, J. Touch, H. Kruse, S. Ostermann, K. Scott, and J. Semke. Ongoing TCP Research Related to Satellites. RFC 2760, February 2000.

[4] H. Balakrishnan, V. Padmanabhan, S. Seshan, S. Stemm, and R. Katz. TCP Behavior of a Busy Web Server: Analysis and Improvements. *Proceedings of the Conference on Computer Communications (IEEE Infocom)*, Mar. 1998.

[5] M. Christiansen, K. Jeffay, D. Ott, and F. D. Smith. Tuning RED for Web Traffic. *SIGCOMM Symposium on Communications Architectures and Protocols*, pages 139–150, Sep. 2000.

[6] S. Dawkins, G. Montenegro, M. Kojo, V. Magret, and N. Vaidya. End-to-end Performance Implications of Links with Errors. Internet-draft, work in progress, Jul. 2000.

[7] S. Floyd and V. Jacobson. On Traffic Phase Effects in Packet-Switched Gateways. *Internetworking: Research and Experience*, 3(3):115–156, Sep. 1992.

[8] S. Floyd and V. Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, Aug. 1993.

[9] S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky. An Extension to the Selective Acknowledgement (SACK) Option for TCP. RFC 2883, Jul. 2000.

[10] J. Padhye and S. Floyd. The TBIT Web Site. URL "http://www.aciri.org/tbit/".

[11] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgment Options. RFC 2018, Apr. 1996.

[12] K. K. Ramakrishnan and S. Floyd. A Proposal to add Explicit Congestion Notification (ECN) to IP. RFC 2481, Jan. 1999.

[13] S. Savage, N. Cardwell, D. Wetherall, and T. Anderson. TCP Congestion Control with a Misbehaving Receiver. *ACM Computer Communication Review*, 29(5):71–78, Oct. 1999.