The Effectiveness of End-to-End Congestion Control Mechanisms

J. Bolliger[1], U. Hengartner[1] and Th. Gross[1,2]
{*bolliger, hengartner, thomas.gross*}@*inf.ethz.ch*

[1]Departement Informatik
ETH Zürich
CH 8092 Zürich

[2]School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

ETH Technical Report #313, 1999

Abstract

TCP's success is in part due to its ability to deal with congestion, yet congestion control remains an important topic for today's Internet protocols. Over time, several enhancements have been proposed that improve TCP Reno's congestion control mechanism, e.g., FACK TCP, Rate-Halving, and TCP Vegas. Most of these enhancements are motivated and evaluated by simulations or small-scale experiments. To assess the effectiveness of these improvements in practice, we performed an Internet experiment during six months using a set of hosts in North America and Europe. We measured protocol dynamics of about 25,000 bulk-data transfers using various congestion control mechanisms.

Although this study is limited in that it provides only a snapshot of a small part of the rapidly changing Internet, it allows us to draw the following conclusions when comparing the proposed TCP enhancements to the baseline Reno protocol: Overall, SACK-enhanced protocols are more robust against packet loss (for loss rates smaller than 10 %). As a consequence, they achieve significantly higher bandwidths. Furthermore, these protocols utilize the available network resources more efficiently as they cause fewer unnecessary retransmissions. The global nature of our experiment allows us to see that the benefit of SACK-enhanced protocols varies widely, e.g., it is almost a factor two bigger for intra-continental than for inter-continental connections. Vegas-style congestion window management during the congestion avoidance phase moderately improves on the number of (multiple) packet loss events, but it does so at the expense of lower throughput.

These findings are supported by protocol micro-measurements: SACK-enhanced protocols (i) are highly effective in avoiding timeouts due to burst losses and due to non-trigger of recovery and (ii) are able to detect and repair lost (fast) retransmissions. Furthermore, we show that Rate Halving's congestion control strategy is a win in situations with small congestion windows, because it is able to evoke additional duplicate acknowledgments and thus reduces the number of non-trigger timeouts. Nonetheless, non-trigger timeouts still account for a significant fraction of the timeouts experienced by Rate Halving. Although Rate-Halving's retransmission strategy is more conservative than FACK's, overall, Rate-Halving coupled with lost retransmission detection experiences fewer timeouts.

1 Introduction

Congestion is a serious problem for today's wide-area networks. Congestion is bad for users and applications, but it is also bad for the network: when a packet encounters congestion, there is a good chance that the packet is dropped, and the dropped packet wasted precious network bandwidth along the path from its sender to its untimely death.

TCP's success as the most widely used Internet transport protocol is due, in part, to its ability to deal with congestion. Nonetheless, various studies identified shortcomings of TCP's congestion control mechanisms and proposed techniques to improve both data recovery and congestion control. These improvements have been evaluated mainly with simulations and in some cases with restricted field tests. However, to date there has been no large-scale comparative evaluation of the suggested TCP enhancements. Although simulations are a valuable tool to understand protocol dynamics and the interaction of various protocols, no amount of simulation can provide a satisfactory assessment of the effectiveness of a protocol, given a complex and rapidly changing environment such as the Internet.

To obtain a better understanding of the various techniques suggested for data recovery, congestion avoidance, and congestion recovery, we implemented several promising techniques (TCP Vegas [8], FACK TCP [23], Rate-Halving [24], and Lost Retransmission Detection [24]) and performed an in-vivo evaluation in the Internet. This paper describes the experiment and reports how the above enhancements compare against the "classical" congestion control scheme included in TCP Reno [32].

The remainder of the paper is organized as follows: Section 2 introduces congestion control in TCP Reno and some of the enhancements proposed for it. Section 3 presents related work in the area of protocol evaluation. The protocol framework developed to evaluate these protocol enhancements is described in Section 4. The evaluation methodology and the detailed setup of the experiements are outlined in Section 5 and Section 6. Section 7 provides an overview of the evaluations performed. Sections 8–12 present all the evaluations in turn. Finally, Section 13 summarizes our findings.

2 End-to-end congestion control

TCP Reno is the most widely used transport protocol today, and although its designers paid great attention to the behavior under congestion, various improvements have been suggested. In this section, we briefly review congestion control in TCP Reno, some of its weaknesses, and some of the enhancements proposed.

2.1 Congestion control in TCP Reno

TCP Reno maintains a so-called "congestion window" (*cwnd*) and tries to keep its size proportional to the available bandwidth. Since today's networks do not provide explicit feedback about the currently available bandwidth to an end-point, the protocol itself has to estimate this value by exploiting implicit feedback from the network, such as the time between sending a packet and receiving an acknowledgment for it. While sending data, the amount of data outstanding (i.e., unacknowledged by the receiver) must never exceed the size of the congestion window (and the size of the buffer available at the receiver). To determine and adapt the size of the congestion window to the bandwidth available, TCP Reno operates in three phases: slow-start [17], congestion avoidance [17], and congestion recovery [18, 31].

At start-up, the congestion window is set to the size of one packet. During the following slow-start phase, the congestion window is exponentially opened to allow the sender to quickly increase the send rate to find the available bandwidth. Unless packet loss occurs, the congestion window is opened until the slow-start threshold value *ssthresh* is reached, at which point the opening is slowed down (i.e., it becomes linear) and the congestion avoidance phase (i.e., the steady state) is entered. In this phase, TCP attempts to send packets at the maximum rate that avoids packet loss and to find out about opportunities for additional data transmission. That's why, even in steady state, TCP slowly attempts to increase the send rate.

Acknowledgments in TCP Reno are cumulative, that is, they acknowledge the reception of all packets up to the sequence number contained in the acknowledgment. In case of receiving an out-of-order packet, a TCP receiver cannot generate an acknowledgment for this packet, instead the receiver has to send another acknowledgment for the latest received in-order packet, a so-called "duplicate acknowledgment". After having received three such duplicate acknowledgments, a TCP sender assumes that the next packet to be acknowledged got lost and enters the congestion recovery phase[1]: The "fast retransmit" algorithm [18,

[1] Note that both packet loss and packet re-orderings make a receiver generate duplicate acknowledgments. Since packet re-

31] retransmits the lost packet ("fast retransmission"). On sending the fast retransmission, the congestion window is halved to allow the network to recover from the congestion that is assumed to have caused the packet loss. The "fast recovery" algorithm [18, 31], which sets in after the fast retransmission, attempts to maintain TCP's self-clock [17]. It does so by estimating the amount of data that remains outstanding in the network and hence by controlling the amount of new data that is allowed into the network ("packet conservation principle"). When TCP has recovered from the data loss (i.e., when the lost packet has been acknowledged), it leaves the congestion recovery phase and enters the linear window growth regime of congestion avoidance.

Since the loss of duplicate acknowledgments or a small congestion window can prevent a sender from triggering a fast retransmission, TCP Reno additionally sets a timeout on every transmitted packet. If a packet remains unacknowledged for the duration of the timeout period, the packet is retransmitted, *ssthresh* is set to half of the current size of the congestion window (i.e., *ssthresh* now contains a "safe" value), which itself is then set to the size of one packet, and a slow-start is performed. Therefore, the bandwidth is drastically reduced in case of a timeout. This effect is aggravated by having timeout values in the range of seconds to cope with variations of the time between a packet is sent and the arrival of its acknowledgment (which very often lies in the range of milli-seconds). These settings cause large gaps between the loss of a packet and the realization of a timeout, and during this gap, no data is sent at all.

2.2 TCP shortcomings and enhancements

We categorize the shortcomings in TCP's congestion control and data recovery mechanisms and the remedies proposed in the literature according to the congestion control phase affected.

2.2.1 Initial slow-start

The static initial slow-start threshold very often leads to overshooting the available bandwidth during the exponential opening of the congestion window. This overshooting frequently results in multiple packet loss. Two approaches that address the problem of overshooting *ssthresh* have been proposed:

- Hoe [16] tries to avoid this overshooting by estimating the bandwidth available to a connection and hence the initial value of *ssthresh* based on round-trip time measurements of the first few packets sent.

- TCP Vegas [8] opens *cwnd* more conservatively and tries to anticipate congestion (see below).

2.2.2 Congestion avoidance

TCP Reno's non-adaptive "linear increase" mechanism in the congestion avoidance phase results in periodic self-induced packet loss. TCP Vegas [8] attempts to avoid the occurrence of this phenomenon by adapting the congestion window to the capacity of the network-path; the capacity is deduced from delay measurements (of this network-path). As a consequence, TCP Vegas opens the congestion window only if additional capacity becomes available and reduces the congestion window in case of incipient congestion.

2.2.3 Congestion recovery: Shortcomings

The fast retransmit algorithm is optimized for the case of a single packet loss in a window. Several studies have shown that TCP Reno has performance problems when multiple packets are dropped from a single window [12, 16, 11]. Figure 1 illustrates the problems with sequence plots of a 1 MB transfer from Linz

orderings should not result in a fast retransmission, TCP assumes that a packet got lost only after having received three duplicate acknowledgments.

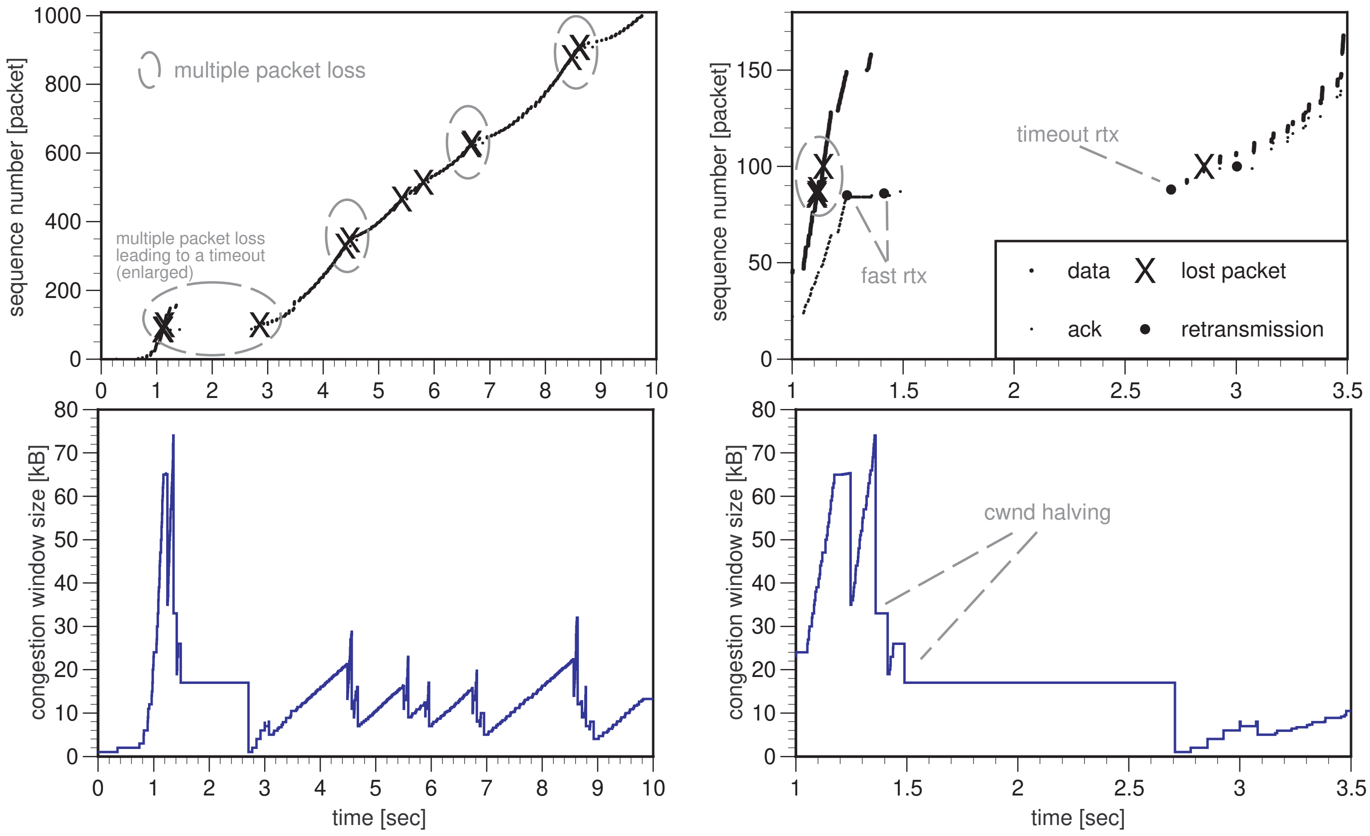


Figure 1: Multiple loss scenario (transfer Linz – Munich)

(Austria) to Munich (Germany). A sequence plot shows the sequence number of every packet transmitted and the time of its transmission by the sender. It also contains the sequence number reported by an acknowledgment and the time of its reception.

The protocol used for the transfer in Figure 1 is our user-level implementation of TCP Reno (see Section 4). The figures on the left side show the entire transfer. The figures on the right side zoom in on the first multiple loss event. The top figures show the sequence plots, where dropped packets are marked with "X" and multiple losses are encircled. The lower figures depict the size of the congestion window during the course of the connection. Note how the packet losses match the reductions of the congestion window size.

Packets 85, 86, and 88 are dropped from the same window of data. These losses lead to two fast retransmit/fast recovery cycles and to two halvings of the congestion window. With TCP Reno, only one loss can be recovered per round-trip time [11][2]. Because of this limitation, and since the congestion window is reduced once per packet dropped, the third packet loss can no longer be recovered by a fast retransmit and a timeout has to be awaited (at $t = 2.75s$).

To sum up, TCP Reno suffers from the following two problems in case of multiple packet loss:

Data recovery Reno's data recovery algorithm allows to retransmit only one dropped packet per round-trip time.

Multiple window halvings Instead of taking the burst loss as the congestion signal upon which to reduce

[2] A fundamental consequence of the exclusive use of cumulative acknowledgments (i.e., the absence of selective acknowledgments) is that the sender has to choose from two alternatives to recover from lost data [11]: (i) retransmitting at most one dropped packet per round-trip time (TCP Reno), or (ii) retransmitting packets that might have already been successfully delivered (TCP Tahoe).

the congestion window, Reno's congestion control treats each packet from the burst of packets dropped as an independent congestion signal resulting in multiple congestion window halvings.

The following three problems cannot be seen directly from Figure 1, but are also inherent to TCP Reno:

Data in flight TCP Reno assumes that all the packets with sequence numbers higher than the one of the fast retransmission are still in flight upon entering a recovery and it considers the arrival of a duplicate acknowledgment a signal for such a packet having left the network and hence as an opportunity to send new data. Clearly, in case of multiple packet loss, but also in case of lost duplicate acknowledgments, TCP overestimates the number of packets still in flight, which inhibits the sending of new data.

Non-trigger Lost duplicate acknowledgments or multiple packet loss (which inhibits the generation of duplicate acknowledgments) may prevent TCP from triggering a fast retransmission for a lost packet, and the loss has to be fixed by a timeout.

Repeated loss The loss of a fast retransmission always leads to a timeout.

2.2.4 Congestion recovery: Enhancements

Several remedies to TCP's shortcomings have been proposed in recent years, which are briefly reviewed here:

Data recovery The use of selective acknowledgments (SACKs) [25], which not only cumulatively acknowledge packets up to a certain sequence number, but also report packets received out-of-order, allows to retransmit more than one dropped packet per round-trip time: Instead of transmitting new data during congestion recovery, TCP implementations exploiting the information provided by SACKs (e.g., SACK TCP [11] and FACK TCP [23]) first retransmit not yet selectively acknowledged data. New data is transmitted only if all packets in any such "hole" have already been retransmitted.

Rate-Halving [24], an enhancement of FACK TCP, also retransmits not yet selectively acknowledged data, but achieves a clearer distinction between data and congestion recovery: For a packet in a "hole" to become eligible for a retransmission, three SACKs selectively acknowledging packets with higher sequence numbers than the one of the candidate for a retransmission have to arrive. Thus, Rate-Halving applies the logic used for triggering a fast retransmission to trigger retransmissions of packets in a "hole".

Multiple window halvings Observations based on Hoe's work [16] suggest a simple fix of TCP Reno, which treats multiple packet loss as a single congestion signal to avoid multiple reductions of the congestion window: NewReno [14], SACK TCP, and FACK TCP leave recovery only after all packets sent before the recovery have been acknowledged. Note that NewReno, as opposed to the TCP versions using SACKs, is not able to retransmit more than one packet in a round-trip time: After retransmitting a packet, it has to wait for a partial acknowledgment (i.e., an acknowledgment for some, but not all of the data outstanding at the beginning of a recovery) to find out which packet (if any) has to be retransmitted next.

Data in flight The forward acknowledgment (FACK) [23] algorithm exploits the information present in SACKs to better estimate the amount of data outstanding in the network during the congestion recovery phase, and thus allows to (re)transmit (new) data earlier and to better maintain TCP's self clock: FACK considers all packets with lower sequence numbers than the highest sequence number selectively acknowledged having left the network (except for retransmissions). Only packets with higher sequence number and retransmissions are assumed to be in flight. Therefore, FACK's estimation of the amount of data outstanding is robust in the face of multiple packet loss and the loss of duplicate acknowledgements, as long as the duplicate acknowledgements received by the sender convey the highest sequence number of all out-of-order packets received by the receiver.

Non-trigger The information provided by SACKs can be exploited to enhance the condition to trigger fast retransmit/fast recovery: FACK TCP additionally enters the recovery phase when the difference between the highest sequence number selectively acknowledged and the sequence number of the next data packet to be acknowledged is larger than two packets. Note that in case of a single packet loss and no losses of duplicate acknowledgments, this condition is identical to the original condition, which triggers recovery after receiving three duplicate acknowledgements. Again, this enhancement increases TCP's robustness in the face of the loss of duplicate acknowledgments and multiple packet loss.

Repeated loss SACKs allow to circumvent the problem of lost retransmissions: Rate-Halving associates every retransmitted packet with the sequence number of the next data packet to be sent containing new data. As soon as a packet whose sequence number exceeds this cached value is acknowledged, and the retransmitted packet has not been (selectively) acknowledged, Rate-Halving assumes that the retransmission has also been lost and schedules the packet for another retransmission.

2.3 Goal

This paper attempts to evaluate some of the enhancements in a real-life setting. One purpose of such an experiment is that it allows us to confirm (or challenge) earlier simulations or measurements. The Internet continues to change, and simulations are unlikely to capture the mix of fast and slow links that is encountered by an actual transfer. Reliance on historical data alone may lead to incorrect conclusions. Another forward-looking benefit is that an understanding of the effectiveness and shortcomings of the proposed enhancements may direct protocol developers to those areas that exhibit opportunities for further improvements. Additional to the evaluation of the enhancements, the amount of data collected also allows us to draw general conclusions about the state (e.g., packet loss rate) of the Internet during our experiments.

3 Related work

The enhancements to TCP's congestion control mechanisms proposed by Lin et al. [22] and Balakrishnan et al. [4] have not been included in our study, because they were published after our experiments had started. Apart from the simulations and experiments conducted by the designers of the various protocol enhancements, there are only a few independent studies that try to evaluate the effectiveness of these enhancements. Some protocol enhancements, such as Rate-Halving, have not yet been evaluated thoroughly by live experiments, and most others have mainly been tested in a restricted environment.

In a simulation study, Fall et al. [11] show how SACKs allow TCP to deal with multiple packet loss and retransmit more than one packet in a round-trip time. Bruyeron et al. [9] compare SACK TCP to TCP Reno, both in a lab testbed and over two Internet paths. For the latter case, SACK TCP is reported to achieve throughput achievements ranged between 15% and 45%.

Simulations are performed by the inventors of the FACK algorithm to evaluate the effectiveness of FACK TCP's improved estimation of the data currently in-flight [23]. FACK TCP is considered less bursty than SACK TCP and superior to SACK TCP in phases of heavy loss.

Based on WAN emulations and live experiments between two fixed Internet hosts in North America, Ahn et al. [1] found that TCP Vegas has a smaller and less fluctuating mean RTT, causes fewer retransmissions, and achieves higher throughputs than TCP Reno.

Hoe [16] pursues the same evaluation strategy to demonstrate that TCP Reno obtains better performance during slow-start by dynamically estimating *ssthresh* and treating partial acknowledgments as indication of lost segments.

4 Protocol framework

To evaluate some of the TCP enhancements presented in Section 2.2, we need both an implementation of these enhancements and an environment to evaluate them. This section deals with the first issue. Section 5 addresses the second issue.

An implementation of the protocols for the purpose of a large-scale comparative study based on Internet experiments has to account for the following difficulties: portability, ease of deployment, and flexibility in selecting and parametrizing the protocol to be used for an experiment.

4.1 User-level protocols

Protocols like TCP have been implemented in user space before [10, 33], but usually efficiency concerns motivate a kernel implementation. A user-level implementation may involve additional copies from user to kernel space, or may constrain the applications on where received data can be kept. Since this experiment aims at investigating the effectiveness of protocols in controlling congestion (i.e., a network property), host-side concerns are of secondary importance (since we can take appropriate action to make sure that host-side behavior does not influence our measurements). Even though, kernel implementations may require fewer copy operations, our user-level implementation is capable of delivering competitive performance (see Appendix A.4). To allow a study that involves many hosts, we cannot rely on OS changes, and we must run our protocol in user mode.

The major motivation for implementing the protocols in user space, however, is that the experiment must allow an external selection of the protocol to be used. Our base transport protocol includes the congestion control mechanisms of TCP Reno, but differs in some aspects from TCP (see Appendix A for a detailed discussion), however, these differences do not affect congestion control. The most important aspect is that our transport protocol is *packet*-streaming (and not *byte*-streaming as TCP). Therefore we use the term *packets* instead of *segments* in this paper.

Our transport protocols run on top of UDP instead of directly on top of IP, and therefore they must live with the restricted information provided by UDP; for example, no ICMP Source Quenches. However, none of these disadvantages affects the results of this study, and placing the protocol in user mode made the wide distribution of our evaluation system possible.

4.2 Flexible, dynamically parameterisable protocol implementation

To ensure that our experiments measure properties of the protocols, we have to be sure that all protocols are implemented in the same way. As stated in Section 2.1, TCP operates in three phases: slow-start, congestion avoidance, and congestion recovery. Protocol enhancements proposed in the literature often address a specific problem that can be localized in/attributed to one of these phases. So to maximize code reuse (or code sharing), we derive as much code as possible from a common code base (i.e., our implementation of TCP Reno). Then each enhancement can be viewed as selecting a different "strategy" for the corresponding phase (i.e., an application of the strategy pattern [15]). For example, TCP Vegas describes an alternative to linearly increasing the congestion window in the congestion avoidance phase. Additionally, by splitting the congestion recovery phase into three building blocks (i.e., actions to be performed upon entering, being in, and leaving a recovery phase), we can easily replace the default behavior of TCP Reno by, for example, NewReno's new way of leaving a recovery, or FACK TCP's extended condition to enter a recovery.

5 Experimental setup

5.1 Control of experiments

Our goal is the evaluation of TCP protocol enhancements. The continuously changing nature of the Internet presents us with the problem that we are unable to repeat a test under the same conditions with a different protocol. However, we can obtain a picture of the average behavior of some path by performing the same experiment several times for every protocol type. But such data collection presents us with another problem: to find a typical Internet path that can be used for our tests. Even if such a path exists, it may be impossible to find it. Observing a large number of different network paths with differing characteristics (e.g., short vs. long round-trip times, high vs. low loss rates, etc.) — instead of observing just a single path — at least reduces the risk that conclusions about protocol-related properties are fooled by specific properties of the underlying network path.

Therefore, we recruited a number of Internet sites and distributed our protocol implementation and a simple application that can issue/participate in a transfer of n bytes length using a specified protocol p with a set of parameters s. (Section 6.3 provides details on the parameters n, p, s used in our experiment). To operatively control the experiment, we adopted Paxson's measurement framework, consisting of the network probe daemon (NPD), which is installed at each participating site, and a control program, which is run on our local workstation [30, 28]. At randomly chosen moments, the control program then chooses two sites (a source and a sink) and a protocol p, contacts the NPDs at both sites and asks them to carry out a probe, that is, a transfer of n bytes using the protocol p specified. The transfer is monitored as described below and the traces are sent back to the controller for further analysis. The measurement intervals, that is the intervals between measurements of the same connections, are taken from an exponential distribution. Therefore, the measurements correspond to additive random sampling. This kind of sampling is unbiased because it samples all instantaneous signal values with equal probability. Paxson [30] explains that with such a sampling, asymptotically, the proportion of the measurements that observe a given state should be equal to the amount of time that the Internet spends in that state. And since we choose the protocol to be used by a given measurement probe randomly from a uniform distribution, we can easily compare protocols based on the measurements collected by comparing the respective distributions.

Furthermore, note that due to possible asymmetries of the routes or the properties of the shared network path between two hosts [30], N hosts allow us to observe $N \times (N-1)$ different network paths. Therefore, a fairly modest number of participating sites allows to measure the dynamics of a reasonable number of different network paths.

5.2 Data collection

There exist two options for the design of a monitoring and analysis system that can provide detailed information. With off-line analysis (outside the protocol stack), the system records a lot of events that are subsequently interpreted. *tcpdump* [20] is a well-known example of this approach, and although it can provide a wealth of data without perturbing the protocol operations, it is extremely difficult to reconstruct information about protocol internals (such as the number of timeouts, size of the congestion window, the packets that are dropped, etc.) from the packet traces [30]. The problems are exacerbated if the monitoring of the connection is done at a different host on the same LAN as the host running the protocol stack to be analyzed [30], but such an arrangement is desirable to reduce perturbations caused by the monitoring.

On the other hand, on-line monitoring and analysis (within the protocol stack) have the advantage of having all the relevant information about protocol internals readily available (i.e., there is almost no additional computation needed to infer relevant information). Furthermore, this approach is easily extended to capture those aspects of the protocol dynamics that are relevant for a particular study. However, active monitoring and on-line analysis consume CPU resources and therefore may perturb the results slightly.

Since our experiment configuration requires only a moderate amount of data to be logged, we chose the second option. To minimize the impact of data collection, our implementation takes care to avoid unnecessary I/O so that the processing overhead stays within acceptable bounds. Another reason that influenced our decision for the second approach is that it can be extended easily to support network-aware applications [6], which need up-to-date information about the status of the network (and the protocol) at run-time.

5.3 Limitations

This paper merely presents a snapshot of six months of measurements; without doubt, the Internet has changed by now. Furthermore, we used a limited number of host pairs that may not be representative for the totality of Internet paths. However, measurements from this restricted group of hosts is more informative than just observing one path, using a dedicated network, or simulations alone.

Another issue that we cannot address is the *fairness* of a protocol. We cannot assess the influence a large-scale deployment of one TCP enhancement may have on other connections or on the Internet as a whole. An investigation of this aspect remains future work, and given the complexity of such an investigation, simulations may provide the most practical approach.

We have every reason to believe that our conclusions carry over to TCP. However, since we started with a new implementation of the protocol and the enhancements discussed in this paper, we cannot comment on the difficulty to fit these enhancements into one of the more popular TCP implementations.

6 Experiments

6.1 Protocols implemented

From the enhancements listed in Section 2.2, we chose the most promising ones (i.e., TCP Vegas, FACK TCP, and Rate-Halving) and added them to our user-level implementation of TCP Reno. In the case of TCP Vegas, we implemented only its modified behavior during congestion avoidance, and we omitted the proposed changes to slow-start and congestion recovery. This omission allows us to more clearly evaluate the effects of the first change.

We refer to the various protocols in this paper as follows:

Reno: Baseline, TCP Reno congestion control;

Forward acknowledgment (FACK): SACK TCP with congestion recovery based on the forward acknowledgment algorithm[3];

Rate-Halving (RH): Retransmission strategy that waits for three SACKs for each of the packets lost, congestion recovery strategy that avoids the instantaneous halving of *cwnd* on entering recovery and instead continually lowers *cwnd*, such that it is halved after one RTT;

Lost Retransmission Detection (LRD): Rate-Halving plus detection and repair of lost retransmissions;

Vegas: TCP Reno's slow-start and congestion recovery phase combined with a TCP Vegas-style congestion avoidance phase (i.e., RTT measurement, estimation of available bandwidth, and *cwnd* adaptation).

6.2 Participating hosts

The sites recruited for our in-vivo Internet experiment are listed in Table 1. From every site, at least one host participated in our experiments. For this report, we exclude campus-wide connections and concentrate on protocol behavior in wide area networks.

[3] Note that our SACKs acknowledge packets instead of blocks of bytes, see Appendix A.2 for details.

Host domain	Machine type	Operating system	Location
North America			
lcs.mit.edu	Sun Sparc	Solaris 2.5	Cambridge, MA
ics.uci.edu	Sun Sparc	Solaris 2.5	Irvine, CA
uwaterloo.ca	DEC Alpha	Digital UNIX 3.2	Waterloo, Canada
research.digital.com	DEC Alpha	Digital UNIX 3.2	Palo Alto, CA
cs.cmu.edu	DEC Alpha	Digital UNIX 4.0	Pittsburgh, PA
Europe			
inf.ethz.ch	SGI	IRIX 6.2	Zurich, Switzerland
unibe.ch	Sun Sparc	Solaris 2.5	Bern, Switzerland
cie.uva.es	Sun Sparc	Solaris 2.5	Valladolid, Spain
ssw.uni-linz.ac.at	Sun Sparc	Solaris 2.5	Linz, Austria
abo.fi	Sun Sparc	Solaris 2.6	Turku, Finland
e-technik.tu-muenchen.de	DEC Alpha	Digital UNIX 4.0	Munich, Germany

Table 1: Participating hosts.

6.3 Parameters

The NPDs at the sites listed are contacted by the central controller at random intervals. The measurement times are chosen such that each network path between two of the NPD sites is probed with a mean interval of 10 hours and that the inter-measurement intervals obey an exponential distribution (i.e., on average, each site participates once per hour in a measurement).

Each probe is characterized by the tuple (p, n, s), where p specifies the protocol to be used[4], n the number of bytes to be transferred, and s the set of protocol-specific parameters to be used. The protocol to be used for a particular probe was chosen randomly with equal probability for each of the protocols[5]. We chose n to be 1 MB and s to specify *ssthresh* as 64 kB (which is the default value for current TCP implementations), packet size as 1024 bytes, and receive buffers of 256 kB, which means that we used large windows to avoid the performance of the transfer being receiver window limited. The values for n and s guarantee that even for connections experiencing no loss at all, a significant part of a connection is spent in congestion avoidance, and Vegas is given a chance to adapt the congestion window.

During August and September of 1997, we debugged and refined our measurement system, and we started collecting logs in earnest in October 1997. The data presented in this paper are based on experiments that took place between October 31, 1997 and May 2, 1998.

6.4 Data collected

To avoid overwhelming the reader with data by looking at every possible connection, we summarize the data into four groups: traffic inside (North) America ($A \rightarrow A$), from America to Europe ($A \rightarrow E$), from Europe to America ($E \rightarrow A$), and inside Europe ($E \rightarrow E$). This clustering is quite intuitive and has already been applied by other researchers [30]. Our evaluations have shown this clustering to be reasonable, since we found considerable variations between the four classes of connections. Additionally, we also present the summarized data for all connections. Table 2 summarizes the number of connections logged, clustered by continents.

[4] $p \in \{Reno, Vegas, FACK, RH, LRD\}$.

[5] To be precise: A bug in the configuration of the measurement system resulted in the protocols Reno and Vegas being chosen half as often as the other protocols during the first 1.5 months of our experiment, which is reflected by the total number of connections for each protocol presented in Table 2.

	Reno	FACK	RH	LRD	Vegas	
$A \rightarrow A$	667	858	947	940	693	4105
$A \rightarrow E$	919	1206	1269	1282	914	5590
$E \rightarrow A$	1388	1720	1688	1861	1309	7966
$E \rightarrow E$	1174	1631	1665	1628	1184	7282
All	4148	5415	5569	5711	4100	24943

Table 2: Clustering of connections.

7 Evaluation

7.1 Analysis of protocol dynamics

There are several aspects to the effectiveness of a protocol (or protocol enhancement). Questions about the effectiveness can focus on

- the performance for a single connection, or
- the degree of fairness extended towards all other connections that share a bottleneck.

Since we perform our measurements in today's Internet, we can modify the parameters of only a single connection; all other connections are beyond our control. So our measurements may help only with understanding the first issue.

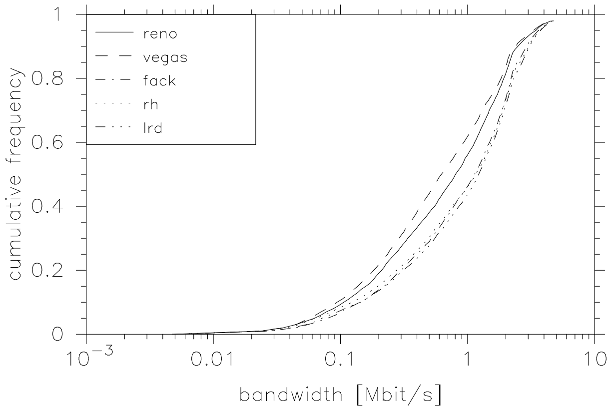
Therefore, our experiments present an extensive set of micro performance data of protocols (e.g., number of timeouts, number of recoveries, number of lost packets). These data are interesting from two points of view: they provide an evaluation of various protocol enhancements in a real-life setting, and they may provide insight to protocol designers.

7.2 Evaluations performed

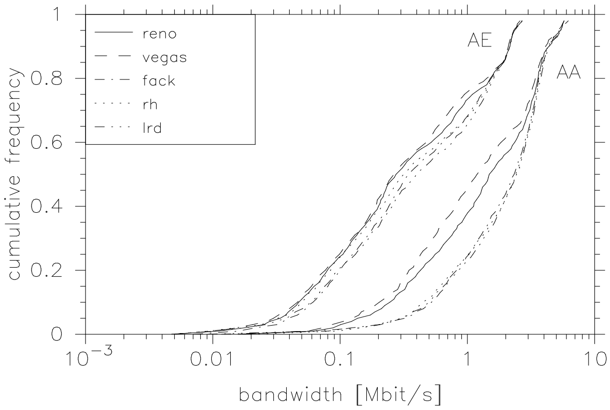
The presentation of the results proceeds as follows: Section 8 presents the net effect of the different enhancements by comparing the throughputs of the five protocols. Moreover, we analyze the protocols' robustness against packet loss and look at how much of the bottleneck bandwidth is utilized by each protocol variant. In Section 9, we turn to the question of how effectively the protocols use the bandwidth available to them by looking at their goodputs. By comparing protocol internal parameters, such as the number of timeouts in Section 10, we investigate whether and how the changes in achieved throughput or goodput can be explained by changes in the protocol internal parameters and whether the proposed protocol enhancements achieve their objectives (as they were primarily targeted at metrics such as avoiding packet loss, timeouts, etc.). Such a comparison of protocol internal parameters across different protocols must be put in relation to the loss rate experienced by the connections of different protocols. Loss events are examined in Section 11. Section 12 concludes our evaluations by tackling the question of how much redundancy in the selective acknowledgements is needed or beneficial.

8 Bandwidth

If — from an application perspective — Vegas, FACK, RH, and LRD are to be improvements to Reno, they should deliver some benefits to an application that utilizes them. A possible benefit is an increase of the bandwidth provided to the application (throughput). Therefore, we first investigate throughput-related metrics for the protocols under consideration.



(a) All connections.



(b) Geographical clustering.

Figure 2: Absolute bandwidths (CDF).

8.1 Absolute bandwidths

Figure 2 (a) shows the cumulative distribution functions (CDF) of the achieved bandwidth for all connections. Note that this figure (and all other CDF plots presented in this paper) has a log-scaled x-axis. The more a graph (distribution) is shifted to the right, the more bandwidth is achieved by the corresponding protocol. In terms of application-level throughput, Vegas performs consistently worse than Reno (for Reno, the median is 0.80 Mbit/s, as opposed to 0.62 Mbit/s for Vegas). Moreover, the SACK-based protocols FACK, RH, and LRD clearly outperform the other two protocols (they achieve median bandwidths of 1.13, 1.16, and 1.22 Mbit/s). We also note that LRD's ability to recognize lost retransmissions results in slightly higher throughput.

Studying continent-to-continent variations (Figure 2 (b) contains the CDFs for $A \to A$ and $A \to E$ connections) not only shows that transatlantic connections achieve significantly lower throughputs than intra-continent connections (this observation also holds for $E \to E$ and $E \to A$ connections), but also that the differences between the protocols are much more pronounced (almost a factor of 2) for the intra-continental connections than for inter-continental connections. The (absolute) difference between the geographic clusters can be explained by the differences in the available bandwidths, that is, intra-continent connections provide more bandwidth to a single connection than transatlantic connections.

The (relative) difference between the protocols can be explained as follows: The throughput of a connection achieved by Reno and its SACK-based enhancements is proportional to

$$\tilde{B} = \frac{1}{RTT\sqrt{p}}$$

where RTT corresponds to the average round-trip time of the connection and p to the loss indication probability, which is defined as the ratio of the number of loss events (timeouts + recoveries) to the number of packets sent [26]. The two curves in Figure 3 plot $\tilde{B}$ for two fixed values of RTT, that is, for the average round-trip time of all $A \to A$ connections ($RTT = 0.10s$) and all $A \to E$ connections ($RTT = 0.25s$) respectively.

For each curve, we mark $\tilde{B}$ for two fixed values of p: the average measured loss indication probability p for Reno and FACK in the two regional clusters $A \to A$ and $A \to E$. Due to its smaller loss indication probability p and its smaller average round-trip time RTT, FACK achieves a higher bandwidth improvement for $A \to A$ than for $A \to E$ connections, which corresponds to the observation we made in Figure 2 (b). Thus we conclude that the benefit of SACK-enhanced protocols is bigger for low-loss, low-RTT connections

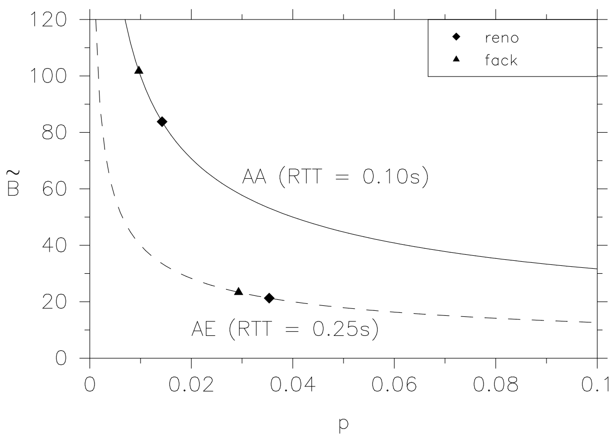


Figure 3: Relative bandwidth for AA and AE connections.

than for high-loss or high-RTT connections. Section 8.2.2 confirms this statement: there we see how the bandwidths achieved by the different protocols in our experiment vary depending on the loss rate, and we show that the bandwidths are nearly identical for high loss rates, regardless of the protocol used.

8.2 Normalized bandwidth

To analyze the effectiveness of a particular protocol enhancement in relevant situations (e.g., for connections that experience multiple packet loss), we would like to compare the effective bandwidths achieved using protocol 1 and protocol 2. However, every network path exhibits different characteristics, so we can at most compare the performance of protocols on connections between two given hosts A and B[6]. However, if we do this, we end up with a very small number of connections and bandwidth measurements which prevent a meaningful comparison. Although there are about 5,000 bulk data transfers for every protocol, there are about 100 host-pairs, and of course only a fraction of the connections exhibits the property that we want to investigate (e.g., multiple packet loss).

However, simply clustering the data, for example, based on location of the hosts, is no solution, since such a clustering ignores the differences in the various network-path characteristics. Given the range of hosts, the paths can exhibit tremendously different properties, and these differences introduce a high degree of variation, which does not stem from the protocol characteristics under consideration and may make it difficult to support any protocol-specific conclusions.

A solution is the normalization of the achieved bandwidth of each transfer to its estimated bottleneck bandwidth. The bottleneck bandwidth is the maximum forwarding rate of the slowest element in the end-to-end chain of network elements (links or routers). It is calculated by applying Paxson's "packet bunch mode"-algorithm for robust bottleneck bandwidth estimation [29, 30] to the traces recording data packet departure/arrival.

The normalization with respect to bottleneck bandwidth proved tedious but turned out to be quite beneficial and informative. For example, it allowed us to identify a high variation of the achieved bandwidth on some connections between the US and Switzerland that we could ultimately trace back to an upgrade in the network infrastructure (a 2 MBit/s link was replaced by an 8 MBit/s multi-pipe link). The bottleneck bandwidths measured range from T1 (1.5 MBit/s) to Ethernet (10 MBit/s).

[6] Even this assumption does not necessarily hold, since the route between the two hosts can change and different connections might experience different link characteristics.

8.2.1 Correlation with loss rate

As earlier studies indicated, there is a strong negative correlation between the data loss rate of TCP Reno and its achieved throughput [30]. The results from Section 8.1 show that FACK, RH, and LRD achieve higher absolute bandwidths than Reno and Vegas. The SACK-based protocols therefore seem to hold the promise of better coping with multiple packet loss and hence better utilizing the bandwidth available. We thus expect the correlation between the throughput and the data loss rate to be smaller for FACK, RH, and LRD than for Reno.

To determine how the throughput of a connection correlates with the connection's data loss rate, we applied the methodology described by Paxson [30]: We compute the correlation between the logarithm of the normalized throughput and the data loss rate, where the logarithmic transformation is to reduce the otherwise dominating effect of throughput outliers.

The correlations for Reno, FACK, RH, and LRD are -0.73, -0.62, -0.69, and -0.57 respectively. With a value of -0.73, the correlation for Reno is even more pronounced than in Paxson's study [30], which mentions a correlation of only -0.52. We also note that FACK, RH, and LRD connections indeed have smaller correlations between bandwidth and loss rate.

8.2.2 Robustness against packet loss

The results of Section 8.2.1 may be interpreted as an indication of the robustness of the different protocols in the face of packet loss: From the correlations between throughput and loss rate, a summary metric, we can derive that the throughput of SACK-enhanced protocols depends less strongly on the ratio of packet loss than is the case for Reno. Thus, based on this summary metric, we may conclude that the SACK-enhanced protocols are more robust against packet losses.

To gain more insight, we try to characterize a protocol's performance (i.e., throughput) as a function of the loss rate experienced. According to a study performed in a testbed environment [9], SACK TCP and TCP Reno achieve identical performance in situations with no packet loss and in situations with high packet loss, (i.e., loss rates exceeding 9% for single packet loss and 3% for burst losses). If there is no loss, TCP Reno and SACK TCP have exactly the same behavior (by design). If there is high loss, the congestion window remains small and timeouts cannot be avoided, even with SACK TCP. This seems to indicate that the robustness of SACK-enhanced protocols against packet loss seems to depend strongly on the actual loss rate experienced. Our data — gathered in a real-world environment — allows us to verify this statement and to include additional protocols (e.g., Vegas).

We compute the loss rate for every connection and cluster the connections in bins of size 0.02. For every bin, the median bandwidth is determined. Figure 4 then plots the median bandwidth as a function of the loss rate experienced for each protocol enhancement. A value of x on the x-axis represents all connections with loss rate $(x - 0.02, x]$. We only show the results for Reno, Vegas, and FACK. The results for RH and LRD connections are very similar to those for FACK connections.

Confirming the results of the previous study [9], we find that for connections with no packet loss, Reno and FACK achieve identical throughput. Furthermore, high-loss environments (loss rate > 0.1) have an equally bad effect on Reno and FACK performance. However, for medium-loss environments (loss rate $<$ 0.1), the SACK-enhanced protocols achieve considerably (49–89%) higher throughputs and therefore seem to be more robust in the face of packet loss than Reno.

Looking at the results for Vegas connections, we note that for low loss connections (0.02 $\leq$ loss rate $\leq$ 0.04) Vegas achieves slightly lower throughputs than Reno. For other loss rates, no marked differences between the two protocols are discernible. This may be interpreted as follows: In our experiment, Vegas and Reno only differ in the way they update/adapt the congestion window during the congestion avoidance phase. In contrast to Reno's strict linear increase, Vegas tries to "sense" incipient congestion and it proactively tries to avoid packet loss by reducing the sending rate (i.e., the congestion window). As will be

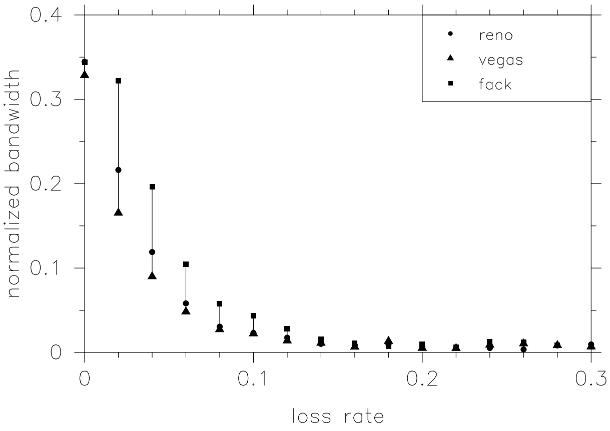


Figure 4: Normalized bandwidth (median) as a function of loss rate.

explained in Section 11, Vegas is indeed (moderately) successful in doing so (for low-loss connections). But as our data indicates, it does so at the expense of performance, that is, it adapts the congestion window too conservatively. This convervatism is best seen for connections experiencing no loss at all: Reno connections keep increasing the congestion window, whereas Vegas connections may keep the window constant or even decrease its size during some periods. Therefore, even for no loss at all, Reno achieves higher bandwidths than Vegas.

8.2.3 Utilization of bottleneck link

As mentioned in Section 7.1, our methodology does not allow us to control other connections that share the bottleneck link with one of our connections. Therefore, we are unable to determine the influence of, for example, a FACK connection on a Reno connection. We can determine only the ratio of the bottleneck link bandwidth consumed by one of our connections. According to the results in Section 8.1, the SACK-enhanced protocols deliver higher bandwidths and we thus expect them to consume a higher ratio of the bottleneck bandwidth than Reno (and Vegas).

Figure 5 plots the CDFs of the normalized bandwidths for all protocols. FACK, RH, and LRD indeed achieve a higher utilization of the bottleneck link than Reno, which itself achieves a higher value than Vegas. Overall, the percentage of the bottleneck bandwidth used by one of our protocols is quite low; in 50% of the cases, FACK consumes at most 24% of the bottleneck bandwidth and the corresponding figures are even lower for Reno and Vegas.

Our data do not allow us to conclude whether the additional bandwidth consumed by the SACK-enhanced protocols was taken away from other connections sharing the bottleneck link or whether it was achieved by better exploiting the available bandwidth. The former would be harmful in the case of a deployment of TCP implementations using SACKs in the Internet, as allowed by a current Internet Draft [2]: during the transition period, in which “old” TCP variants share links with SACK-enabled TCP versions, the SACK versions would take all bandwidth, while Reno connections would starve.

However, a study conducted in a testbed points out that this additional bandwidth is not taken away from TCP Reno [9]: both TCP Reno and SACK-enhanced TCPs adhere to the same principles of congestion control, that is, linear increase and multiplicative decrease [17]. The observed difference in bandwidth consumed results from SACK TCP’s better use of the bandwidth wasted by TCP Reno: TCP Reno unnecessarily treats multiple packet loss in a single congestion window as multiple congestion signals and therefore gives bandwidth away by unnecessary reductions of the congestion window which may even lead to timeouts.

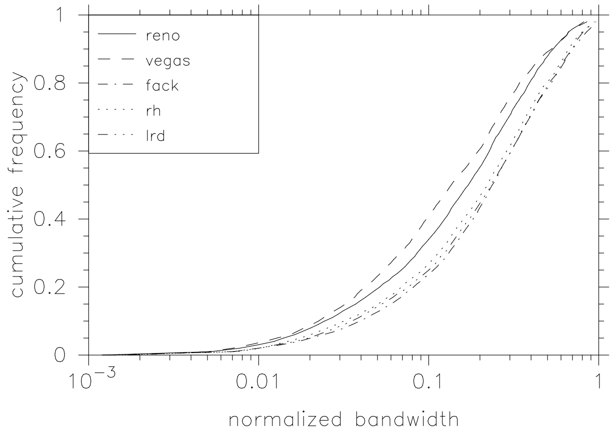


Figure 5: Bandwidths normalized to bottleneck bandwidth(CDF).

9 Goodput

An application wants its data to be transmitted as fast as possible. From the network's viewpoint, however, it is important to minimize the number of data packets required to reliably transmit the application data to prevent a waste of resources. The metric of interest for the application is throughput. For the network-centric viewpoint, the metric of interest is goodput, where goodput is defined as the ratio of the amount of data generated by the application to the amount of data transmitted over the network. Obviously, a protocol that succeeds in keeping the goodput close to one is superior to a protocol that achieves the same throughput, but that must transfer more data.

Section 8.1 has shown that FACK, RH, and LRD achieve higher bandwidths than Reno and Vegas. We now investigate whether they must generate more data packets to do so. Figure 6 plots the CDFs of the goodput for Reno, Vegas, RH, and LRD[7]. Note that the x-axis starts at 0.6. For low goodputs ($<$ 0.90), the distribution is nearly identical for all protocols. The same applies to high goodputs ($>$ 0.99). For goodput values between these two limits, RH and LRD connections on average achieve higher goodputs than Reno and Vegas connections.

As will be shown in Section 11, all protocols experience similar loss rates. Since Reno and Vegas achieve lower goodputs than RH and LRD, they must evoke more unnecessary retransmissions than the SACK-enhanced protocols. The source of such unnecessary retransmissions are slow-starts following timeouts due to multiple packet loss: Every acknowledgment triggers the retransmission of two packets with adjacent sequence numbers. Since the first of these two packets has not been acknowledged, it is reasonable to assume that its first transmission was dropped. However, this assumption need not hold for the second packet: its first transmission may have arrived at the receiver, but there is no way for the receiver to tell the sender. Section 10 reveals that Reno and Vegas indeed suffer more timeouts due to multiple packet loss than the SACK-enhanced protocols and thus are more likely to generate more unnecessary retransmissions.

To sum up, the SACK-enhanced protocols not only achieve higher throughputs than Reno and Vegas, they also use the available resources more efficiently, that is, they cause fewer unnecessary retransmissions. Our data thereby backs a claim made by Floyd [13], who predicted the number of unnecessary retransmission to be low for SACK TCP.

[7] A bug in the implementation of FACK prevents us from presenting the data for FACK. However, the other evaluations are not affected.

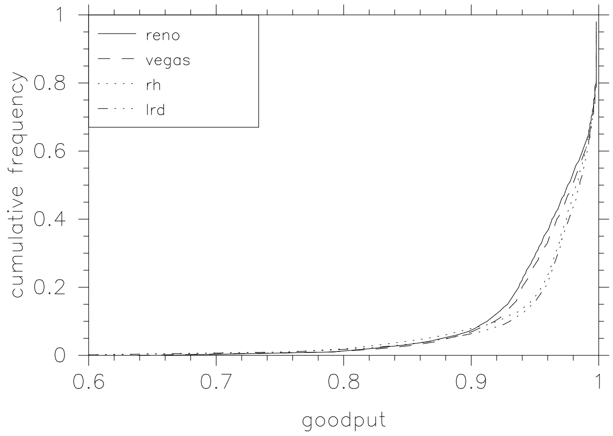


Figure 6: Goodput per connection (CDF).

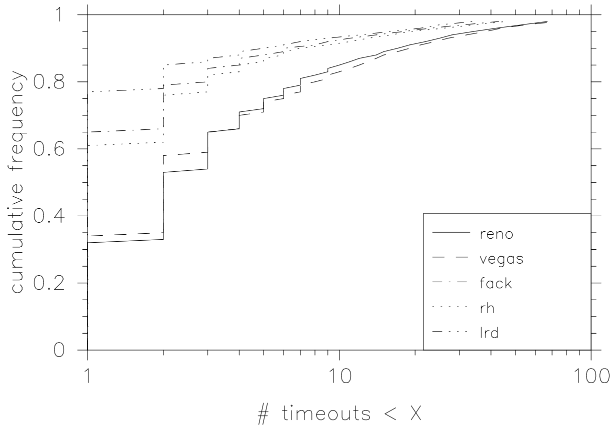


Figure 7: Number of timeouts per connection (CDF).

10 Timeouts

As we have seen in Section 8.1, FACK, LRD, and RH provide more bandwidth to an application than Reno and Vegas. To explain this difference, we take a closer look at some protocol internal parameters, such as the number of timeouts per connection.

The use of selective acknowledgments and FACK's enhanced congestion recovery strategy aim at coping better with multiple packet loss. The expectation is that these improvements reduce the number of timeouts due to such loss events. Although Vegas employs the same congestion recovery algorithms as Reno, its congestion avoidance algorithm, which tries to avoid (multiple) packet loss altogether, should still have a positive effect on the number of timeouts.

10.1 Overall comparison

Figure 7 presents the CDFs of the number of timeouts per connection for Reno, Vegas, FACK, RH, and LRD (logarithmically scaled x-axis). A value of x means $x - 1$ timeouts per connection. The more a graph (distribution) is shifted to the left, the fewer timeouts occur for the corresponding type of connection. The distributions for Reno and Vegas are almost identical. The clear left shifts in the distributions for FACK, LRD, and RH show that these protocols significantly cut down the number of timeouts per connection. 61.0% of the RH, 65.9% of the FACK, and 77.3% of the LRD connections do not suffer any timeouts at all, whereas only 32.8% of the Reno and 34.1% of the Vegas connections are timeout-free.

10.2 Classification

To shed more light on the results reported above, that is, to understand the reasons why FACK, RH, and LRD outperform Reno and Vegas in terms of timeouts, we try to isolate the effects of their enhancements. To do so, we first classify the timeouts according to their causes into the following four categories. A similar clustering has also been suggested by Lin et al. [22].

Multiple packet loss For Reno and Vegas, multiple packet loss leads to multiple halvings of the congestion window. In case of too many losses (both of data packets and acknowledgments) or too small a congestion window, the fast retransmit algorithm cannot be triggered for all lost packets and a timeout is evoked. Since FACK, LRD, and RH treat multiple packet loss as a single congestion signal and therefore remain in recovery until all the data outstanding prior to the recovery has been acknowledged, timeouts due to multiple packet loss always happen in congestion recovery for these protocols. There are two causes for such timeouts:

- **Missing acknowledgments** The flow of acknowledgments triggering (re)transmission of data ebbs off.
- **Lost MPL retransmission** A retransmission distinct from the fast retransmission (i.e., a retransmission caused by multiple packet loss) is dropped (MPL = Multiple Packet Loss).

Non-trigger of recovery Packet loss situations which fail to trigger a fast retransmit/fast recovery because of a small congestion window or high losses of acknowledgments result in so-called non-trigger timeouts. This kind of timeouts is caused either by a single lost packet or by the first packet lost in a multiple packet loss situation (as opposed to the timeouts due to multiple packet loss, which are caused by the second or higher packet lost in a multiple packet loss situation).

Lost fast retransmission None of Reno, Vegas, FACK, and RH are able to recover from a lost fast retransmission without evoking a timeout. Only LRD may become aware of the loss and get around it without triggering a timeout.

Lost timeout retransmission All of the five protocols can recover from the loss of a timeout retransmission only by another timeout.

10.3 Timeouts in Reno

Before turning to a detailed investigation of each protocol enhancement's effectiveness in dealing with multiple packet loss and in avoiding certain types of timeouts, we must first understand their impact on the base protocol (Reno).

The heights of the bars in Figure 8 depict the mean number of timeouts per connection, where the average is based on all connections using a given protocol. Additionally, the figure shows a detailed breakdown of the causes for these timeouts[8]. We find that the majority of timeouts suffered by Reno connections are caused by multiple packet loss (36% of all timeouts) and by non-trigger of a recovery (41% of all timeouts). This finding ties in with results reported by Balakrishnan et al. [4]. Lin et al. [22], however, report different results: More than 85% of the TCP timeouts in their study are due to non-trigger, whereas only 11% are

[8] For RH and LRD, we present only the data for the second three months because of a flaw in the computation of the timeout interval in the case of timeouts on lost fast and lost timeout retransmissions during the first three months. A detailed analysis of the results for the other protocols reveals that the numbers are nearly identical regardless of which three-month period is considered, which indicates that it is reasonable to make such a comparison. Note that all other evaluations are based on the data for the complete period of six months, as the effect of this flaw is negligible for these enhancements.

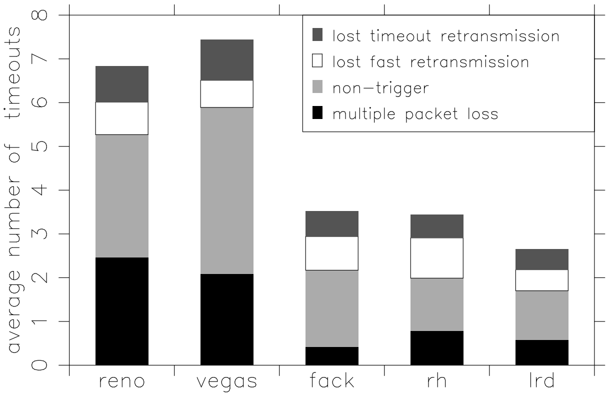


Figure 8: Breakdown of timeout causes for each protocol.

due to multiple packet loss. Lost fast retransmissions and lost timeout retransmissions account for 11% and 12% of our Reno timeouts respectively.

We now turn to investigating each class of timeouts in the context of the four protocol enhancements and to determining how effective each enhancement is in reducing the kind of timeouts it claims to reduce. The results for Reno we just presented thereby serve as baseline.

10.4 Multiple packet loss

As shown in Figure 8, Vegas succeeds in reducing the number of timeouts due to multiple packet loss by 15% compared to Reno. The reason for this improvement is Vegas's modified management of the congestion window during congestion avoidance. With the help of this modification, Vegas seems to be able to avoid multiple packet loss and suffers fewer such timeouts.

By handling multiple packet losses in a window in a single recovery, FACK is highly effective in cutting down the number of timeouts due to such multiple packet losses (reduction by 83.0% compared to Reno). Almost all the Reno connections that suffered at least one timeout (67.2%) also suffered from at least one timeout due to multiple packet loss (65.4%). However, only about half of the FACK connections that suffer a timeout (34.1%) also suffer from a multiple loss timeout (18.3%). RH and LRD also succeed in reducing the number of timeouts due to multiple packet loss, but not as drastically as FACK (RH: 68.5%, LRD: 77.0%).

To understand the differences among the SACK-enhanced protocols as far as their effectiveness in dealing with multiple packet loss situations is concerned, we briefly review the relevant mechanisms: FACK and RH/LRD employ different *congestion control strategies* during recovery. FACK halves the congestion window on entry of fast retransmit. The first opportunity to (re)transmit data is after about half an RTT. After that period, data can be (re)transmitted for every incoming duplicate acknowledgment ("dup"). RH/LRD's congestion strategy tries to continually lower the congestion window so that it is halved after one RTT. Data can be (re)transmitted for every other dup directly after entering recovery. FACK and RH/LRD also differ in the *retransmission strategy* used. FACK retransmits not yet selectively acknowledged data immediately, whereas with RH/LRD, a not yet selectively acknowledged packet becomes eligible for retransmission only after three dups have arrived that selectively acknowledge packets with higher sequence numbers.

We can now turn to analyzing the effect of these congestion control and retransmission strategies on the two types of multiple loss timeouts identified in Section 10.2. Figure 9 provides a breakdown of the number of multiple loss timeouts per connection reported in Figure 8 according to the two possible causes for such a timeout, that is, missing acknowledgments and lost MPL retransmissions.

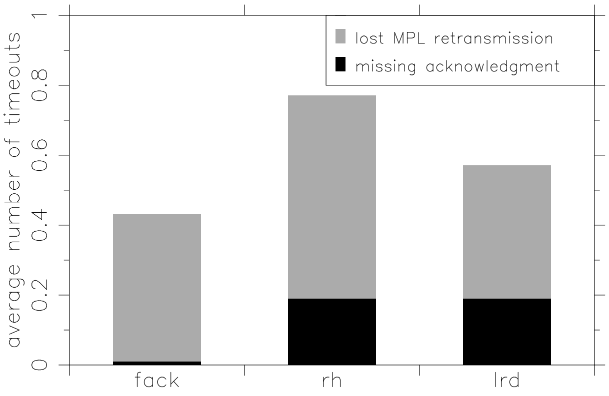


Figure 9: Breakdown of causes for multiple packet loss timeouts.

10.4.1 Missing acknowledgments

The number of timeouts of this class of timeouts may serve as an indicator of how well a protocol can maintain the self-clock in situations with multiple packet loss. At first sight, the results are somewhat counter-intuitive: FACK, whose instantaneous halving of the congestion window may actually disrupt a connection's self-clock, very rarely experiences timeouts due to missing acknowledgments. On average, there are only 0.01 such timeouts per FACK connection, whereas RH and LRD, which could — in principle — continually (re)transmit data, experience 0.19 such timeouts. This is explained as follows: the retransmission strategy dominates the effects of the congestion control strategy. The only packet that may cause such a timeout in FACK is the one following the packet acknowledged by the first partial acknowledgment in a recovery[9], as this is the only packet whose retransmission can be prevented by the congestion control strategy. Further partial acknowledgments always cause one or more (re-)transmissions. With RH/LRD, candidates for retransmission must wait for three SACKs, which increases the risk of timeouts due to missing acknowledgments. Based on these observations, the following two questions must be answered:

Is FACK's retransmission strategy too aggressive? FACK's strategy of retransmitting not yet selectively acknowledged packets immediately may be too aggressive, for example, in situations with packet reorderings. However, our data reveals that not yet selectively acknowledged packets are very rarely retransmitted unnecessarily, that is, shortly before the SACK arrives: 74% of all connections with at least one recovery experience zero, 9% one, and only 17% of all these connections experience two or more such unnecessary retransmissions. This implies that the blocking for half an RTT seems to be quite effective in avoiding unnecessary retransmissions.

Can RH's retransmission strategy be improved? A possible remedy might be the addition of FACK's extended condition to enter recovery to RH and LRD's retransmission algorithm: A packet is also retransmitted if the difference between its sequence number and the highest selectively acknowledged packet becomes larger than two packets. By determining this difference for every such timeout, we can estimate the fraction of such timeouts that might have been avoidable with this extended retransmission trigger mechanism. We find that for RH, only 15.7% of this class of timeouts have a difference larger than two packets (LRD: 21.0%). Note that these numbers serve as an upper bound on the reduction potential, since RH and LRD's congestion control might not have allowed the sending of a retransmission in all the cases.

10.4.2 Lost MPL retransmission

Overall, lost MPL retransmissions account for most of the timeouts due to multiple packet loss for all the SACK-enhanced protocols. Such timeouts make up 11.8% of all FACK timeouts. 18% of all connections

[9] A partial acknowledgment is an acknowledgment for some, but not all of the data outstanding at the beginning of the recovery.

are affected and could therefore benefit from mechanisms to detect and repair such lost retransmissions.

In principle, timeouts due to lost MPL retransmissions should be equally likely for FACK and RH. LRD should be able to improve on the number of such timeouts due to its ability to recognize lost retransmissions.

Again, the results are somewhat surprising: RH suffers 28% more timeouts caused by lost MPL retransmissions than FACK. Our data do not allow us to pin down the cause for this discrepancy, but the following statement may provide a reasonable explanation: Multiple packet loss is a sign of "severe" (i.e., non-negligible) congestion in the network. Waiting half a RTT before resuming (re)transmission — the model employed by Reno's and FACK's congestion control strategy — gives the network time to recover from its congested state and thus makes it less likely that subsequent (re)transmissions are lost. RH's strategy of continually (re)transmitting data, although at half the original rate, risks to hit the congestion period as it trails off and thus risks that its retransmissions are dropped. This phenomenon remains subject to further investigation.

Furthermore, Figure 9 shows that LRD suffers 34.5% fewer timeouts due to lost MPL retransmissions than RH and about the same number of such timeouts as FACK. This implies that LRD is really able (and quite effectively so) to detect and repair lost MPL retransmissions (and to compensate for the increase in such timeouts witnessed by RH).

10.5 Non-trigger of recovery

Vegas According to Figure 8, Vegas causes 34.7% more timeouts due to non-trigger of a recovery than Reno. This increase is a consequence of the rather conservative congestion window adjustment strategy implemented by Vegas, whose negative influence on bandwidth has already been mentioned in Section 8.2.2. The more conservative the window adjustment is, the smaller is the average congestion window and the more likely it is that non-trigger timeouts occur.

FACK FACK's extended recovery trigger condition is responsible for the decrease in the number of non-trigger timeouts (37.5% improvement over Reno). When looking at each recovery, we note that they were triggered by three, two, and one duplicate acknowledgment in 58.3%, 26.5%, and 12.0% of the cases.

RH There are fewer non-trigger timeouts for RH than for FACK. At first look, this decrease might seem astonishing because RH's condition for sending a fast retransmission requires the arrival of three packets selectively acknowledging packets with higher sequence numbers, whereas FACK may trigger a fast retransmission after as early as after one SACK, provided the distance between the highest selectively acknowledged packet and the packet for which a cumulative acknowledgment is awaited is bigger than two packets. However, when taking a closer look at this kind of timeouts for FACK, we note that the average size of the congestion window before such a non-trigger timeout is 3.63 packets. Therefore, on average, at most three packets are in transit before the timeout: the first one being lost and the other two packets resulting in at most two duplicate acknowledgments. But there is no way for these two acknowledgments to trigger a recovery (and a fast retransmission) in the case of FACK, since the distance mentioned above cannot exceed three packets. In contrast, RH already enters a recovery after having received the first duplicate acknowledgment and triggers the transmission of a data packet for every second duplicate acknowledgment: The first duplicate acknowledgment triggers the transmission of a new data packet. The third duplicate acknowledgment results in the fast retransmission. Assuming that both of the duplicate acknowledgments mentioned above arrive, we are sure to have a transmission, which can evoke another duplicate acknowledgment. With the help of this acknowledgment, the sender will finally be able to trigger the fast retransmission.

The hypothesis that RH is able to survive a loss in a congestion window having a size of fewer than four packets is confirmed by the following figure: Only 13.9% of the recoveries that had a congestion window smaller than four packets on entry resulted in a timeout. In the remaining 86.1% of the cases, RH was able to trigger a fast retransmission.

LRD LRD suffers nearly the same average number of non-trigger timeouts as RH, which is not astonishing, as both protocols apply the same mechanism to trigger fast retransmissions.

Even though the SACK-based protocols are able to cut down the number of non-trigger timeouts compared to Reno, all the protocols still experience a considerable number of such timeouts. Non-trigger timeouts are likely to happen in situations with severe congestion, and we have seen in Section 8.2.2 that none of the protocol enhancements studied here are able to cope with such situations. Balakrishnan et al. [4] and Lin et al. [22] both proposed a solution that mitigates the problem of packet loss for connections with tiny congestion windows: They proposed to send new data for every duplicate acknowledgment, which could then generate further duplicate acknowlegements and trigger recovery. Their method is similar to the mechanism used by RH, however, it may be more aggressive than RH, which starts halving the sending rate right away.

10.6 Lost fast retransmissions

Figure 8 shows that LRD succeeds in detecting some of the lost fast retransmissions: it reduces the average number of these timeouts per connection by 48% (compared to RH). Overall, it achieves the lowest number of timeouts on lost fast retransmissions for all five protocols.

The timeouts on lost fast retransmissions reported in Figure 8 allow us to check the plausibility of our experimental setup and the conclusions drawn from the analysis of the experiment. Since the probability that a fast retransmission is dropped should be identical for Reno, Vegas, FACK, and RH (as no lost retransmission detection is performed), the ratio of the number of timeouts on lost fast retransmissions to the number of fast retransmissions should stay constant for those protocols. This fraction is 0.058 for both Reno and Vegas and only slightly higher for FACK (0.066) — an indication of plausibility. However, this does not apply to RH, where the corresponding value is 0.080. We already noticed a similar phenomenon in Section 10.4, where RH has been found to suffer more timeouts due to lost retransmissions in a recovery than FACK.

10.7 Lost timeout retransmissions

Although the SACK-enhanced protocols do not employ mechanisms to avoid timeouts due to lost timeout retransmissions, we can see from Figure 8 that FACK, RH, and LRD experience 30, 36, and 44% fewer such timeouts than Reno. This decrease can be explained as follows:

The SACK-enhanced protocols are effective in reducing the number of timeouts due to multiple packet loss, non-trigger, and lost fast retransmissions (as shown in Sections 10.4, 10.5, and 10.6). Therefore, there are fewer timeout retransmissions. Assuming that the probability for such a timeout retransmission to get lost is identical for all protocols, there are also fewer lost timeout retransmissions and thus fewer timeouts caused by such a loss.

10.8 Discussion

10.8.1 Summary

Our data indicate that SACK-enhanced protocols succeed to drastically reduce the average number of timeouts per connection compared to Reno and Vegas (similar findings are reported by Bruyeron et al. [9]). FACK, RH, and LRD achieve reductions in the average number of timeouts of 48, 50, and 61% compared to Reno.

We note that these results contradict earlier studies [4, 22], which predicted the optimization potential of the usage of selective acknowledgments to be small. Balakrishnan et al. [4] determined only 4.43% of the TCP timeouts to be avoidable by the usage of SACKs. These differences might be attributed to the fact that the connections we traced observed bandwidths that are (at least) an order of magnitude higher than the bandwidth available to the connections in the earlier studies[10]. This change may reflect the general trend towards higher bandwidths but may also be due to the fact that the sites participating in our study are generally well connected to the Internet.

When comparing this section's results on timeouts with the throughput results obtained in Section 8.1, we find correspondence between the bandwidth and timeout results: FACK, RH, and LRD's ability to reduce the overall number of timeouts results in higher throughput, and the fact that Vegas suffers a significantly higher number of non-trigger timeouts is reflected by the lower throughputs achieved (compared to Reno).

10.8.2 Reduction potential in Reno

Two interesting questions from a protocol designer's point of view are: Can we postulate an upper bound of the Reno timeouts avoidable with the help of SACKs, and how well is this reduction potential exploited by those SACK-enhanced protocols that we investigated in this study?

To answer the first question, we assume that the 36% of multiple packet loss timeouts (in Reno) and the 11% of the timeouts due to lost fast retransmissions are avoidable by SACK-enhanced protocols. Furthermore, to get a better handle on the potential benefits of SACK-enhanced protocols in reducing the number of non-trigger timeouts, we address this question: "How many of these timeouts suffered by Reno connections would have been avoidable with FACK's extended condition to trigger a recovery?" For this purpose, we installed a "SACK receiver" (i.e., a receiver generating SACKs for out-of-order packets) at the sink of the bulk data transfer and made the sender trace the incoming SACKs, but not evaluate them. For all non-trigger timeouts, the highest selectively acknowledged packet and the sequence number of the next data packet to be acknowledged at the time of the timeout are determined. This information then allows us to find out whether the extended recovery condition would have triggered a recovery[11]. We find that with the help of SACKs, 29.4% of Reno's non-trigger timeouts or 11.7% of all Reno timeouts could have been avoided. Note that FACK reduces the number of non-trigger timeouts by 37.5%.

Based on these observations, we can postulate an upper bound for the potential benefits of SACK-enhanced protocols: approximately 59% of all the Reno timeouts could be avoided by the use of SACKs, that is, the 36% timeouts due to multiple packet loss, the 12% avoidable, non-trigger timeouts, and the 11% timeouts on lost fast retransmissions. Considering the effect described in Section 10.7, we can expect that due to the reduction in the number of these timeouts, SACK-enhanced protocols will also witness fewer timeouts due to lost timeout retransmissions, and we would thus expect that SACK-enhanced protocols can reduce the number of timeouts compared to Reno by at most 64%[12].

When comparing this upper bound to LRD, which avoids the most timeouts of all the SACK-enhanced protocols we evaluated, we see that it is close to the estimated reduction potential: LRD reduces the number of timeouts by 61%.

11 Loss events

11.1 Data loss

Section 10 discussed the effect of the protocols under consideration on the number of timeouts experienced per connection. This section focuses on the analysis of the number of loss events per connection. A loss event designates either a single lost packet or multiple lost packets in a congestion window. The absolute

[10] Bandwidth evaluations for these studies have been performed by Balakrishnan et al. [5] and Paxson [30].

[11] Since we did not trace the reception of selective acknowledgments by Reno senders for connections during the first three months, this evaluation is based only on the data from the second three months.

[12] 64% = 59 (multiple + avoidable non-trigger + lost fast retransmission timeouts) + (1–0.59) $\cdot$ 12 (lost timeout retransmission timeouts)%

	any loss					multiple packet loss				
α percentile	10	25	50	75	90	10	25	50	75	90
Reno	0	1	4	19	40	0	0	2	6	16
Vegas	0	1	3	18	40	0	0	1	5	14
FACK	0	1	5	18	40	0	0	2	6	17
RH	0	1	4	18	42	0	0	2	7	20
LRD	0	1	3	15	36	0	0	2	6	16

Table 3: Number of loss events per connection.

numbers for the timeouts per connection must be put in relation to the number of loss events experienced by the connection to allow for a valid comparison of the protocols. For instance, the impressive reduction in the number of timeouts achieved by FACK compared to Reno is only meaningful if the number of loss events is of the same order for both protocols. By design of the protocols, that is, because FACK and Reno share the same code for slow-start and congestion avoidance, we would expect this to be the case. In contrast, Vegas' congestion avoidance mechanism aims at reducing the number of loss events by continuously adapting the size of the congestion window to the available bandwidth.

Table 3 presents the number of loss events per connection. We classify loss events into two categories: any kind of loss and multiple packet loss in a window. The second group is contained in the first. The table reports the 10-, 25-, 50-, 75- and 90-percentiles of the number of loss events per connection. For example, at least 10% of all connections experience no loss (in fact, almost 20% of all connections are loss-free).

Any loss As seen in Table 3, overall, the number of losses is nearly identical for all the protocols. Up to the first quartile no differences are discernible and the medians differ only slightly. This implies, that the comparisons made in Section 10 are indeed valid. Vegas' success in reducing the number of losses (compared to Reno) is at best fairly modest.

Multiple loss Table 3 also presents the number of multiple loss events per connection. In this case, Vegas performs consistently better than the other protocols. That is, even though Vegas does not seem to be able to reduce the overall number of loss events, it is (modestly) effective in reducing the number of multiple packet loss events.

Multiple packet loss seemed to be the main motivation for the protocol enhancements studied here and initial slow-start overshoot has been identified as one of the major causes for multiple packet loss [16]. We find that this overshooting of the available bandwidth in initial slow-start leads to packet loss in 59% of all the connections. 42% of all the connections experience multiple packet loss in initial slow-start. There is room for enhancements that try to avoid such loss events, for example, by estimating the bandwidth available and setting the initial slow-start threshhold appropriately [16, 27]. The SACK-enhanced protocols can merely try to keep the "damage" small (i.e., to avoid timeouts)[13], but they have no means to avoid the "damage" altogether.

11.2 Ack Loss

The generation of acknowledgments ("acks") depends on the arrival of data packets at the receiver, but not on the current congestion state of the network path from the receiver to the sender. Therefore, the sending of acks does not adapt to congestion in the network, as opposed to the sending of data packets. Ack loss rates thus allow conclusions about the loss rates which would be experienced by a non-adaptive protocol.

[13] 32% of all Reno connections experience a timeout due to multiple packet loss during initial slow-start, whereas only 5% of SACK-enhanced connections are affected by such timeouts.

	ack loss rate		
	TCP	Reno	Δ
$A \rightarrow A$	1.6%	2.2%	+0.6%
$E \rightarrow E$	2.8%	1.5%	-0.7%
$E \rightarrow A$	11.7%	7.1%	-4.6%
$A \rightarrow E$	3.2%	2.1%	-1.1%
all	4.6%	3.6%	-1.0%

Table 4: Ack loss rates for different connection geographies.

	quiescent			loss rate (non-quiescent)		
	TCP	Reno	Δ	TCP	Reno	Δ
$A \rightarrow A$	69%	48%	-21%	4.4%	4.2%	-0.2%
$E \rightarrow E$	58%	34%	-24%	5.9%	2.2%	-3.7%
$E \rightarrow A$	31%	27%	-4%	16.9%	9.7%	-7.2%
$A \rightarrow E$	52%	42%	-10%	6.0%	3.6%	-2.4%
all	52%	36%	-16%	8.7%	5.6%	-3.1%

Table 5: Conditional ack loss rates for different connection geographies.

The ack loss rates of TCP connections have been thoroughly examined in an earlier study [30]. Since this study is based on data gathered in 1994 and 1995, the comparison of its results with ours allows us to draw conclusions about the development of loss rates in the Internet during the last three years.

Table 4 compares the ack loss rates reported in the earlier study ("TCP") to the ack loss rates measured for our Reno connections ("Reno"). With the exception of $A \rightarrow A$ connections, ack loss rates decreased. $A \rightarrow A$ connections now suffer higher loss rates than $E \rightarrow E$ connections. Note that for $E \rightarrow A$ connections, data flows into America, therefore acks flow into Europe. Similarly, for $A \rightarrow E$ connections, ack streams travel from America to Europe. As mentioned in the earlier study, loss rates higher than 5% have serious adverse effect on TCP performance. $E \rightarrow A$ connections still achieve loss rates in this range, but the situation has improved.

Table 5 presents the percentage of quiescent connections as reported in the earlier study and as experienced by our Reno connections. Quiescent connections are connections for which no ack loss occurred at all. Additionally, the ack loss rates for connections suffering at least one ack loss are given. We note that the decrease of the loss rates for this type of connections is more pronounced than in case of all the connections (as shown in Table 4). Thus the percentage of quiescent connections has to be lower, which is confirmed by Table 5.

12 Selective acknowledgments

Since a non-negligible fraction of acknowledgements are dropped in today's Internet, selective acknowledgements have been designed to provide redundancy [25]. Current TCP implementations allow a maximum of three SACK-blocks to be present in a TCP-header. The question is how much redundancy is necessary to achieve a good tradeoff between header space (and hence bandwidth) wasted and robustness against loss of acknowledgements gained? We try to establish a minimum number of selective acknowledgments per acknowledgment required to overcome most of the acknowledgment losses. For that purpose, we consider a selective acknowledgment to be *used* by the sender if the corresponding packet has not yet been selectively acknowledged. The first, second, ..., sixth selective acknowledgment in an acknowledgment are expected to be used by the sender in different proportions: the first one is most likely to be used, the extent in which the other ones are used is smaller and depends on the probability of acknowledgment (burst) loss, which has

	Selective acknowledgment no.					
	1	2	3	4	5	6
$A \rightarrow A$	97.0(4.8)	2.0(3.3)	0.5(1.1)	0.2(0.7)	0.1(0.5)	0.1(0.3)
$A \rightarrow E$	96.9(6.2)	2.2(4.0)	0.5(1.4)	0.2(0.7)	0.1(0.5)	0.1(0.3)
$E \rightarrow A$	92.4(12.3)	4.7(6.3)	1.6(3.2)	0.7(1.9)	0.4(1.2)	0.2(0.9)
$E \rightarrow E$	98.3(3.2)	1.3(2.1)	0.3(0.7)	0.1(0.4)	0.1(0.3)	0.0(0.2)
All	96.0(8.1)	2.6(4.5)	0.8(2.0)	0.3(1.2)	0.2(0.7)	0.1(0.5)

Table 6: Use of selective acknowledgments.

been investigated by Paxson [30][14].

Table 6 shows the mean rate (parentheses contain standard deviation) of use of the n^{th} selective acknowledgment in an acknowledgment[15]. For example, for connections with both sender and receiver in North America, 97% of all the selective acknowledgments that the sender considered as useful were at the first position in an acknowledgment.

The second–sixth selective acknowledgments are rarely used, compared to the first one. However even the sixth is used, and the decision on how many acknowledgments to support involves a tradeoff between space and frequency of usage. With the top three acknowledgments, the overwhelming number of cases are covered.

13 Concluding remarks

In our experiment, we investigated the effectiveness of various congestion control mechanisms that have been proposed as enhancements to TCP Reno.

In summary, we record that SACK-enhanced protocols are a considerable improvement over Reno-style congestion recovery, as they are more robust against packet loss (for loss rates smaller than 10%), and in particular are highly effective in dealing with multiple packet loss situations. The SACK-enhanced protocols may seem to be more aggressive in that they achieve a higher utilization of the bottleneck bandwidth than Reno, however, they are merely more efficient in using their share of the bottleneck bandwidth, that is, they produce fewer unnecessary retransmissions.

These summary results are confirmed by the findings about protocol micro-measurements: Compared to Reno, SACK-enhanced protocols significantly cut down the number of timeouts due to multiple packet loss and due to non-trigger of recovery. Furthermore, Lost Retransmission Detection proved to be quite an effective mechanism in reducing the number of timeouts in recovery that are caused by lost retransmissions. Overall, LRD reduces the number of timeouts by 61% relative to Reno. This result is impressive as we estimate (based on the kind and frequency of Reno timeouts) that 64% of these timeouts can be removed by the enhancements investigated.

As far as differences among the SACK-enhanced protocols are concerned, we note that Rate-Halving suffers more timeouts due to multiple packet loss than FACK, on the other hand, it is able to survive a packet loss in small congestion windows without a timeout. Overall, these two effects cancel out. We also note that non-trigger timeouts still account for a significant fraction of the timeouts experienced by Rate-Halving and thus offer room for improvements.

Vegas-style network path adaptation in congestion avoidance is able to reduce the number of multiple packet losses, and hence the number of timeouts caused by multiple packet loss, by 15%. However, being rather conservative, Vegas causes more non-trigger timeouts (35%). With regard to the delivered bandwidth, the result is an overall performance that is slightly worse than Reno's.

Our experiment has confirmed a number of earlier investigations, but also produced some results that

[14] Note that our SACKs acknowledge packets instead of blocks of bytes, see Appendix A.2 for details.

[15] The values given are overall for FACK, RH, and LRD.

differ from previous work. The improvement of the SACK-enhanced protocols over Reno is more pronounced than has been reported. Packet loss rates have decreased (with the exception of connections having both sender and receiver in North America). Although RH and LRD are closer to an "ideal" congestion control strategy, as they continously (re)transmit data at half the original sending rate, in practice, the cruder response (halving of the congestion window) included in FACK produces better results in situations with multiple packet loss.

Understanding the behaviour of a network as complicated as the Internet is difficult. Both simulations and experiments are a valuable tool. Experiments allow us to focus simulations on those parts of the design space that is problematic in practice. For example, our experiments show that the relative bandwidth improvement obtained by SACK-enhanced protocols is bigger for intra-continental than for inter-continental connections. Therefore, simulating only the first scenario may result in too optimistic conclusions. We hope that others will be able to add to our understanding of the Internet and that, at some time, the best enhancements to TCP's congestion control find their way into the "production" network protocol.

Acknowledgements

We thank those who provided us with an opportunity to use their host(s) as a platform for our experiments: Digital (DEC SRC, Palo Alto): Lance Berc; MIT (Boston): Volker Strumpen; University of Waterloo (Waterloo, Canada): George Labahn; University of California (Irvine): Michael Franz; Abo Akademi (Turku, Finland): Wolfgang Weck; Universität Linz (Linz, Austria): Josef Templ; Technische Universität München (Munich, Germany): Kurt Antreich; Universität Bern (Berne, Switzerland): Gerhard Jäger; Univeridad de Valladolid (Valladolid, Spain): Miguel Revilla.

Daniel Brennwalder contributed to the implementation of Reno and FACK. We thank Hans Domjan and Peter Brandt for their help and appreciate feedback and comments by Peter Steenkiste.

Research sponsored in part by ETH Zuerich Polyprojekt 41-2641.5, and in part by the Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0287. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Advanced Research Projects Agency, Rome Laboratory, or the U.S. Government.

References

[1] J. S. Ahn, P. Danzig, Z. Liu, and L. Yan. Evaluation of TCP Vegas: Emulation and Experiment. In *Proceedings of ACM SIGCOMM '95*, pages 185–195, August 1995.

[2] M. Allman, V. Paxson, and W. Stevens. Internet Draft: TCP Congestion Control, December 1998. Expires June, 1999.

[3] F. Baker. RFC 1812: Requirements for IP Version 4 Routers, June 1995.

[4] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, M. Stemm, and R. H. Katz. TCP Behaviour of a Busy Internet Server: Analysis and Improvements. In *Proceedings of INFOCOM '98*, pages 252–262, San Francisco, CA, March 1998.

[5] H. Balakrishnan, M. Stemm, S. Seshan, and R. H. Katz. Analyzing Stability in Wide-Area Network Performance. In *Proceedings of ACM SIGMETRICS 97*, pages 2–13, June 1997.

[6] J. Bolliger and T. Gross. A Framework-based Approach to the Development of Network-Aware Applications. *IEEE Transactions on Software Engineering*, 25(5):376–390, May 1998.

[7] R. Braden. RFC 1122: Requirements for Internet Hosts—Communications Layers, October 1989.

[8] L. S. Brakmo, S. W. O'Malley, and L. Peterson. TCP Vegas: New Techniques for Congestion Detection and Avoidance. In *Proc. of ACM SIGCOMM '94*, pages 24–35, London, October 1994.

[9] R. Bruyeron, B. Hemon, and L. Zhang. Experimentations with TCP Selective Acknowledgment. *Computer Communication Review*, 28(2):54–77, April 1998.

[10] A. Edwards and S. Muir. Experiences Implementing a High Performance TCP in User-Space. In *Proceedings of ACM SIGCOMM '95*, pages 196–205, September 1995.

[11] K. Fall and S. Floyd. Simulation-based Comparisons of Tahoe, Reno, and SACK TCP. *Computer Communication Review*, 26(3):5–21, July 1996.

[12] S. Floyd. TCP and Successive Fast Retransmits, February 1995. ftp://ftp.ee.lbl.gov/papers/fastretrans.ps.

[13] S. Floyd. Issues of TCP with SACK. Technical report, Lawrence Berkeley National Laboratory, January 1996.

[14] S. Floyd and T. Henderson. Internet Draft: The NewReno Modification to TCP's Fast Recovery Algorithm, December 1998. Expires June, 1999.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Object-Oriented Software*. Addison Wesley, 1995.

[16] J. C. Hoe. Improving the Start-Up Behavior of a Congestion Control Scheme for TCP. In *Proceedings of ACM SIGCOMM '96*, pages 270–280, Stanford, CA, August 1996.

[17] V. Jacobson. Congestion Avoidance and Control. In *Proceedings of ACM SIGCOMM 88*, pages 314–329, August 1988.

[18] V. Jacobson. Fast retransmit, April 1990. Message to the end2end-interest mailing list.

[19] V. Jacobson, R. Braden, and D. Borman. RFC 1323: TCP Extensions for High Performance, May 1992.

[20] V. Jacobson, C. Leres, and S. McCanne. tcpdump, 1989. ftp://ftp.ee.lbl.gov/tcpdump.tar.Z.

[21] P. Karn and C. Partridge. Estimating Round-Trip Times in Reliable Transport Protocols. In *Proceedings of ACM SIGCOMM '87*, pages 2–7, August 1987.

[22] D. Lin and H. T. Kung. TCP Fast Recovery Strategies: Analysis and Improvements. In *Proceedings of INFOCOM '98*, pages 263–271, San Francisco, CA, March 1998.

[23] M. Mathis and J. Mahdavi. Forward Acknowledgement: Refining TCP Congestion Control. In *Proceedings of ACM SIGCOMM '96*, pages 281–292, Stanford, CA, August 1996.

[24] M. Mathis and J. Mahdavi. TCP Rate-Halving with Bounding Parameters. http://www.psc.edu/networking/papers/FACKnotes/current., October 1996.

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

[26] M. Mathis, J. Semke, J. Mahdavi, and T. Ott. The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm. *Computer Communication Review*, 27(3):67–82, July 1997.

[27] V. Padmanabhan and R. H. Katz. TCP Fast Start: A Technique for Speeding Up Web Transfers. In *Proceedings of IEEE Globecomm '98*, November 1998.

[28] V. Paxson. End-to-End Routing Behavior in the Internet. In *Proceedings of ACM SIGCOMM '96*, pages 25–40, August 1996.

[29] V. Paxson. End-to-End Internet Packet Dynamics. In *Proceedings of ACM SIGCOMM '97*, pages 139–152, Sept 1997.

[30] V. Paxson. *Measurements and Analysis of End-to-End Internet Dynamics*. PhD thesis, University of California, Berkeley, April 1997.

[31] W. Stevens. RFC 2001: TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms, January 1997.

[32] W. R. Stevens. *TCP/IP Illustrated, Vol. 1 and Vol. 2*. Addison Wesley, 1993.

[33] C.A. Thekkath, T.D. Nguyen, E. Moy, and E.D. Lazowska. Implementing Network Protocols at User Level. In *Proceedings of ACM SIGCOMM '93*, pages 64–73, September 1993.

A TCP/Reno differences

This section describes the differences between TCP[16] and Reno[17]. They can be divided into three categories: design issues, differences with regard to TCP-related RFCs, and deviations from existing TCP implementations. The latter ones are pointed out only in relation to congestion control issues. At the end, a comparison of bandwidths achieved by TCP and Reno is presented.

A.1 Design issues

Reno is a user-level protocol and relies on UDP and therefore avoids the introduction of a separate checksum calculation.

Reno is not stream-based. Fixed-sized packets are used to transfer both data and control packets. Therefore, the sequence numbers and the window calculations are packet-oriented. For example, in congestion avoidance, an acknowledgment increments the congestion window by $1/cwnd$ (see below). The size of the packets is statically determined, that is, there is no exchange of the sender's and receiver's *MSS* at startup. For the tests, we chose a data load of 1024 bytes. The Reno header additionally requires 17 bytes, so the packets handed over to UDP have a size of 1041 bytes and could have been subject to fragmentation (a host is required to be able to process IP packets up to a size of 576 bytes [7]) Fragmentation has never been observed in connections involving a host at ETHZ. The described setup guarantees reasonably sized data packets and therefore avoids the occurrence of the Silly Window Syndrome.

To simplify the development of the protocol, several features of TCP with no influence on our tests have been omitted; Reno has no PUSH flag or urgent data pointer. Since our main interest for this experiment focuses on bulk data transfers, Nagle algorithm has not been implemented.

The acknowledgment scheme consists of three different types of acknowledgments: normal acknowledgments are delivered on the reception of every second in-order packet, immediate acknowledgments on the reception of an out-of-order packet, and periodic acknowledgments on the absence of a normal or immediate acknowledgment during the last 200 ms[18]. Periodic acknowledgments impede the probing of zero windows by the sender. Since there are no bidirectional data transfers in our tests, piggy-backing is not supported.

A.2 Differences between Reno and various TCP-related RFCs

RFC 1122 [7] requires a TCP sender to slow down transmission on the receipt of a Source Quench ICMP message. UDP does not notify an application when receiving such a message (in violation of the RFC), so there is no possibility for Reno to react to such a message. Fortunately, these messages are quite rare [30], and their use is deprecated to avoid additional data during phases of congestion [3].

Upon the acknowledgment of new data, RFC 2001 [31] allows a TCP sender to open the congestion window by a factor depending on the MSS (and on the current window size during congestion avoidance), but the window cannot depend on the amount of acknowledged data. However, Reno does use the latter amount for this purpose. Therefore, Reno is more aggressive than TCP in the presence of delayed acknowledgments. We chose this behavior for Reno because it is closer to the spirit of Jacobson's landmark paper [17].

In violation of RFC 1122, Reno does not immediately send an acknowledgment after the reception of an out-of-order packet that does not fit into the receiver window[19]. Thanks to the periodic acknowledgments,

[16] With TCP, we designate the transport protocol as described in RFC 1122 [7].

[17] With Reno, we designate our user-level implementation of a transport protocol with TCP Reno congestion control.

[18] Similar to TCP Reno, the generation of periodic acknowledgments relies on a 200 ms timeout timer. Therefore all periodic acknowledgments following a normal or immediate acknowledgment are distributed over a range of 0 ms to 200 ms after this last acknowledgment.

[19] This event can result from a sender being confused about the current state of the receiver. With an immediate acknowledgment, the receiver tries to remedy this situation as quickly as possible to avoid the unnecessary retransmission of further packets.

the sender will nonetheless become aware of the current window position, although it may take longer than with an immediate acknowledgment.

A selective acknowledgment as described in RFC 2018 [25] consists of two sequence numbers, which acknowledge a block of bytes, and every acknowledgment can contain at most three of these acknowledgments (due to space constraints in the TCP header). Because our protocol is packet-streaming, we use a slightly different acknowledgment scheme: a selective acknowledgment consists of a sequence number acknowledging a packet, and every acknowledgment can contain at most six of these acknowledgments.

A.3 Differences between Reno and TCP Reno

The TCP implementation presented by Stevens [32] (TCP Reno) is considered as implementing TCP correctly (with some small flaws) and is therefore used as base for many other implementations [30]. Therefore we point out some differences between TCP Reno and Reno.

For the RTT calculation, we chose a scheme similar to the one proposed in RFC 1323 [19]. This scheme allows the sender to calculate the current RTT more than once in a window. If permitted by Karn's algorithm [21], every second data packet includes a time-stamp in its header and is immediately acknowledged by the receiver. This enhancement can significantly improve the accuracy of the RTT estimator, especially for large windows [19].

TCP's mechanism to check for a packet suffering a timeout is triggered every 500 ms, whereas Reno retransmits a packet as soon as its RTO has expired.

A.4 Bandwidth Comparison

Although Reno has been implemented in an object-oriented manner in C++ and does not include optimizations like TCP's header prediction, its performance is sufficient to allow conclusions based on Reno. The following absolute bandwidths (in Mbit/s) were measured in an experiment conducted in an lightly loaded local-area network; they support our view that Reno provides a platform for the evaluation of other protocols.

	Bandwidth [Mbit/s]	
	Mean	StdDev
TCP	6.748	0.064
Reno	6.776	0.124