TCP HACK: TCP Header Checksum Option to improve Performance over Lossy Links

R. K. Balan, B. P. Lee, K. R. R. Kumar
L. Jacob, W. K. G. Seah, A. L. Ananda
Centre for Internet Research,
School of Computing,
National University of Singapore,
55 Science Drive 2, Singapore 117599
{rajeshkr, leebp, kaleelaz, jacobl, wseah, ananda}@comp.nus.edu.sg

Abstract

In recent years, wireless networks have become increasingly common and an increasing number of devices are communicating with each other over lossy links. Unfortunately, TCP performs poorly over lossy links as it is unable to differentiate the loss due to packet corruption from that due to congestion. In this paper, we present an extension to TCP which enables TCP to distinguish packet corruption from congestion in lossy environments resulting in improved performance. We refer to this extension as the *HeAder ChecKsum option (HACK)*. We implemented our algorithm in the Linux kernel and performed various tests to determine its effectiveness. Our results have shown that HACK performs substantially better than both SACK and NewReno in cases where burst corruptions are frequent. We also found that HACK can co-exist very nicely with SACK and performs even better with SACK enabled.

Keywords: *Protocol Design, Protocol Analysis, Wireless Networks*

1 Introduction

There has been a proliferation in the use of mobile computing in the last few years. More and more devices are talking to each other via lossy links. Lossy environments are characterised by high bit error rates as opposed to wired networks where the bit error rate is very low. They are also usually served by low bandwidth links and experience long delays during handoff periods. As a result, it has become vital that the network protocols used to interconnect these devices understand and operate well in these lossy environments.

The de-facto network protocol stack used for communications is the TCP/IP stack. This stack couples a best effort network layer (IP) with either a reliable (TCP) or an unreliable (UDP) transport layer. The majority of applications on the Internet use the TCP/IP stack as the basis for their transactions.

However, TCP was designed to optimise its performance to deal with packet losses in the network due to congestion [Jac88]. It is unable to determine if a packet loss is due to congestion or corruption of the packet due to errors in the network. As a result, TCP generally performs poorly in lossy environments as it interprets packet corruption as congestion in the network. Thus instead of increasing or, at least, maintaining its sending rate to overcome these errors due to corruption, TCP will decrease its sending rate to reduce, what it perceives as, congestion in the network. This reduction in sending rate results in low throughputs for bulk transfers.

In this paper, we propose a modification to the TCP [RFC793][RFC2581][S94] protocol that allows it to perform better in lossy environments. We base our solution on the premise that when packet corruption occurs, it is more likely that the packet corruption occurs in the data and not the header portion of the packet. This is because the data portion of a packet is usually much larger than the header portion for many applications over typical MTUs. With this knowledge, we have devised an algorithm by which TCP is able to recover these uncorrupted headers and thus determine that packet corruption and not congestion has taken place in the network. TCP can then react appropriately. We do this by introducing two TCP options: the first option is for data packets and contains the 1's-complement 16-bit checksum of the TCP header (and pseudo-IP header) while the second is for ACKs and contains

the sequence number of the TCP segment that was corrupted.

The rest of this paper is organised as follows. We discuss some related work in Section 2, followed by a description of the details and dynamics of our extension to the TCP protocol in Section 3. Section 4 will describe our implementation while Section 5 presents the results of our experiments. We discuss some possible deployment strategies of our protocol in Section 6. Section 7 will detail our future plans and we conclude with a summary in Section 8.

2 Related Work

There has been an incredible number of techniques developed for TCP over the past decade facilitating fast and efficient recovery from packet losses in general.

The fast retransmit algorithm [RFC2581] interprets incoming duplicate acknowledgements as an indication of packet loss and retransmits the packet indicated by the ACKs while avoiding timeouts. However, if two or more packets have been lost from a window, the fast retransmission will not be able to recover the losses without waiting for a timeout. NewReno [Hoe95][Hoe96][RFC2582] introduces the concept of fast retransmission phase, which starts on detection of a packet loss and ends when the receiver acknowledges reception of all data transmitted at the start of the retransmission phase. The sender assumes reception of a partial ACK during the fast retransmission phase as an indication that another packet has been lost within the window, and retransmits it immediately. With the Selective Acknowledgement (SACK) option [RFC2018] enabled, the receiver sends duplicate ACKs containing the segment numbers of the packets it has received. This allows the transmitter to selectively retransmit only lost packets, without retransmitting already SACKed packets.

Packet loss due to corruption is more common over satellite and wireless networks than wired networks and there have been a number of initiatives in tackling this problem.

A common solution is to add Forward Error Correction (FEC) to the data being sent over lossy links. [RFC2488] covers the issues in using FEC to improve the performance of satellite links. The Indirect-TCP (I-TCP) protocol [BB] splits a TCP connection between a fixed and mobile host into two separate connections and hides TCP from the lossy link by using a protocol optimised for lossy links. The SNOOP protocol [BSAK] caches packets at the base station and performs local retransmissions over the lossy link.

The use of Explicit Congestion Notification (ECN) [Flo94][RFC2481] in the TCP/IP protocol enables routers to inform TCP senders about the onset of congestion and may assist in distinguishing packet losses due to congestion and corruption. Other explicit notification schemes include Explicit Loss Notification (ELN) [BPSK] and Explicit Bad State Notification (EBSN) [BKVP].

3 TCP Header Checksum Option

We extended TCP by including two additional TCP options. The first (see Fig. 1) is both an *enabling option* used in SYN segments as well as the *Header Checksum option* used in data segments. When the option is used in a SYN segment, it is an indication that the Header Checksum option can be used once the connection is established (the value of the option field is ignored in this case). When used in data segments, the option field contains the 16 bit 1's complement checksum of the TCP header and the pseudo-IP header. The second option (see Fig. 2) is the *Header Checksum ACK option* which is included in 'special' ACKs generated in response to packet corruption.

Kind=14	Length=4	1's complement checksum of TCP header and pseudo-IP header

Fig. 1: TCP **Header Checksum** option

Kind=15	Length=6	32-bit sequence number of corrupted segment to resend

Fig. 2: TCP **Header Checksum ACK** option

Normally, TCP carries only one checksum, which is for the *entire* TCP segment. If this checksum fails due to packet corruption, the entire segment is discarded. However, in many cases, the headers of the corrupted TCP segment are still recoverable as the corruption might have occurred in the data portion alone. Hence, by adding a separate checksum

for the header portion of the TCP segment, the TCP receiver will be able to check the integrity of the header. By recovering this header, the receiver is able to send a 'special' ACK back to the TCP sender indicating packet corruption. This ACK will contain the sequence number of the corrupted packet in the option field. This ACK is identical to normal ACKs except for the additional option.

We modified the data processing algorithms of the TCP sender and receiver and the ACK processing algorithm of the TCP sender to incorporate our new Header Checksum options, which are explained in the following subsections.

3.1 Modifications to the TCP sender

When sending out data segments, our modified TCP stack first checks if the Header Checksum option has been negotiated. If the option has not been negotiated, the TCP sender proceeds as per normal. Otherwise, it will compute the header checksum for that data segment and place it into the option field

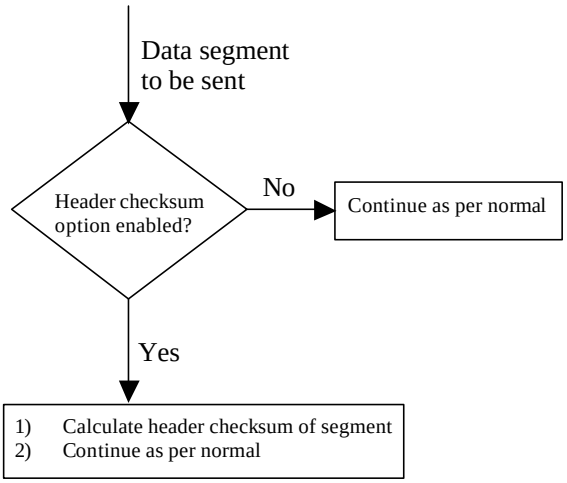

Fig 3. Modifications to the TCP sender

of the Header Checksum option. The rest of the data sending algorithm is as per normal.

3.2 Modifications to the TCP receiver

When the TCP receiver receives a packet, it verifies the integrity of the segment using the standard TCP checksum. If the segment is uncorrupted, it is processed as per normal. However, if it is corrupted, the modified TCP stack does the following:

1) Verify the integrity of the header of the corrupted segment using the value of the header checksum contained in the option field.
2) If the header is corrupted, the segment is discarded and no further processing is done.
3) If the header is intact, the 'special' ACK is sent to the sender of the corrupted packet. This ACK will contain the Header Checksum ACK option indicating to the sender that this ACK was generated in response to packet corruption. It contains the sequence number of the corrupted segment in the option field, thus allowing the sender to selectively retransmit *only* the segment that was corrupted.

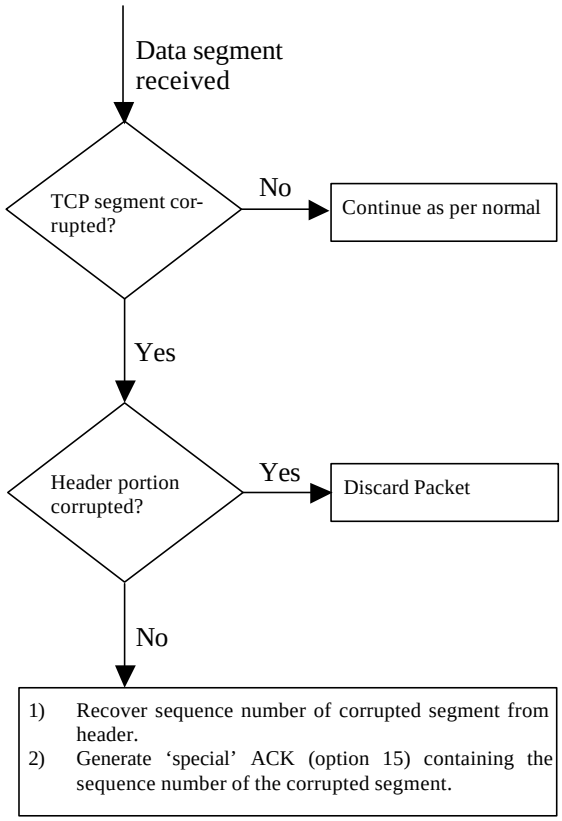

Fig 4. Modifications to the TCP receiver

3.3 Modifications to the ACK processing

When the TCP sender receives an ACK, it checks if the Header Checksum ACK option is present. If the option is not there, it indicates that this is a normal ACK and the sender processes it as per normal. However, if the option field is set, the stack does the following:

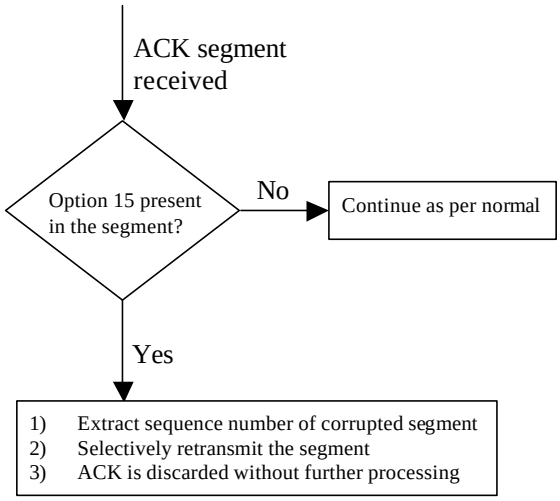

Fig 5. Modification to the ACK processing

1) The sequence number of the corrupted segment triggering this ACK is obtained from the Header Checksum ACK option field.
2) The TCP retransmission algorithm is called to selectively retransmit the corrupted segment. These retransmissions are done at rates permitted by the current congestion window (*cwnd*).
3) No further processing is done unlike the case of normal TCP ACKs.

These 'special' ACKs do not indicate congestion in the network. Hence, the TCP sender does not halve it's cwnd if it receives multiple 'special' ACKs with the same value in the ACK field (for e.g., ACKs generated in response to corruption in consecutive segments. These ACKs will have the same value in the ACK field but different values in the Header Checksum ACK option field).

4 Implementation and Experimental Setup

We incorporated our Header Checksum options and the necessary changes to the TCP algorithm in the Linux kernel version 2.2.10. This modified version of the Linux kernel was installed on our experimental testbed consisting of Celeron 300A machines with 128 megabytes of RAM each. The machines were connected using Intel Ether-Express Pro 100 (set to 10 Mbps) network cards. The experimental testbed is shown in Fig. 6.

We ran our experiments by sending TCP bulk data from the client to the server. We used *iperf* [BC98] to generate this data. The error / delay box was used to corrupt and delay packets in the network to simulate lossy and long latency environments,

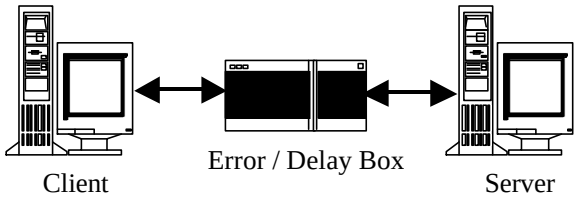

Fig 6. Experimental testbed

respectively. Random as well as bursty packet errors were generated using packet corruption software and the amount and location of the corruptions within a packet were all randomised. For our experiments, errors were generated only to packets travelling on the forward path. Packets on the reverse path (the ACK packets from the server to the client) were not corrupted.

We modified the device drivers of the ethernet cards to stop them from discarding packets that failed the packet CRC checks. As a result, corrupted packets arriving at the network cards were passed up to the TCP/IP stack without being discarded.

5 Results and Discussions

To test the effectiveness of HACK, we ran a variety of test scenarios. These scenarios were designed to test the performance of HACK under various lossy environments. We chose NewReno and SACK for comparison as Linux implements both of them and they are acclaimed as the "best" basic and extended commodity TCP implementations, respectively [BHZ][FF96][RFC2582].

5.1 Random Bit Errors

In the first experiment, we compare the performance of HACK with SACK (on top of NewReno) and NewReno (default TCP stack in Linux 2.2.10) in the presence of white noise, i.e., in a scenario where the error condition is characterized by sharp spikes causing single bit corruptions. We have translated this bit error profile into packet errors *(Our packet corruption probabilities range from 2% to 15% corresponding to a bit error range of $1x10^{-6}$ to $1x10^{-5}$).* We ran the experiment over a low latency link (10 ms end-to-end delay) by sending 2 MB of TCP bulk data from the client to the server and over a long latency link (300 ms end-to-end delay, e.g., satellite link) by sending 256 KB of TCP bulk data from the client to the server. In the

later case, we used 256KB to reduce experiment time as 2 MB was taking too long to complete. We repeated the experiment five times for each TCP/IP stack and for each packet corruption probability. The TCP window size was set high enough that it

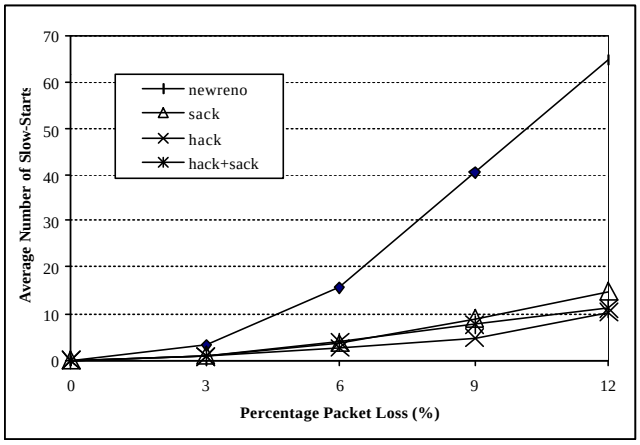

Fig. 7: Average number of slow-starts for long latency link with random single packet errors

was not a limiting factor for either latency. The results of this experiment are shown in Fig. 8 for the low latency link and in Fig. 9 for the long latency link. Fig. 7 shows the average number of slow starts experienced by the various TCP implementations over the long latency link.

As can be seen, both SACK and HACK perform better than NewReno for both latencies. They also experience less slow starts than NewReno. These results were due to the selective ACK feature of SACK (which enabled SACK to do more intelligent and efficient retransmissions of lost packets) and the ability of HACK to recover useful information from corrupted packets. Hence, we exclude NewReno from all further experiments and only show the comparison between SACK and HACK. The performance of both SACK and HACK are comparable in the situation where white noise is prevalent. The best performance is achieved when we combine both HACK and SACK together. This will be discussed further later.

5.2 Burst Errors

We next ran experiments to compare the performance of SACK and HACK in bursty error conditions. Multi-packet random burst errors with burst lengths ranging from 2 to 10 packets were considered. We ran this experiment over the long latency link with the window size set high enough not to be a limiting factor

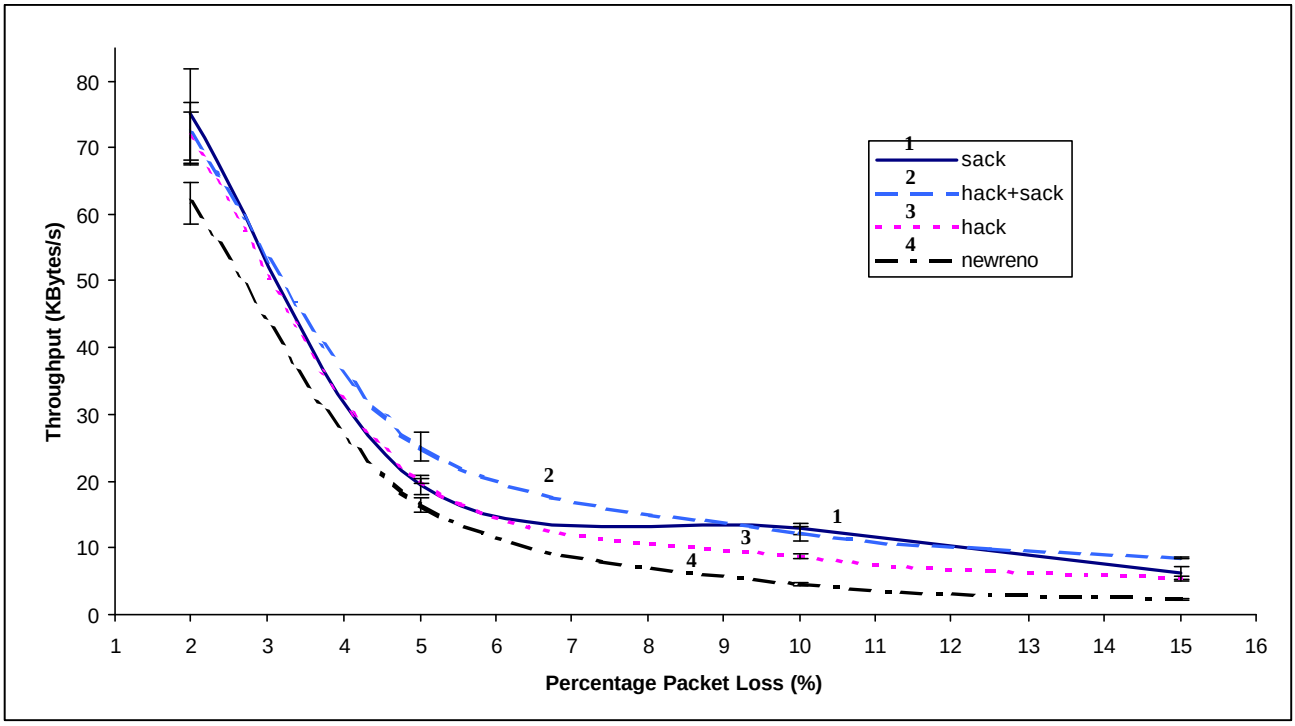

Fig 8. Throughput versus percentage packetlposs for short latency (10 ms) link with random single packet errors

Figs. 10 – 13 show the results of the various TCP schemes under different burst error lengths for 2%, 5%, 10% and 15% burst error probabilities respectively. From the graphs, it can be seen that HACK performs substantially better than SACK in the

presence of bursty errors. This is because SACK is unable to respond when it loses too many packets in a row and thus it times out frequently. HACK is better in this respect as it can recover some of the headers of the corrupted packets and use those headers to generate ACKs and keep the pipe flowing. As expected, HACK performs better with SACK activated than without SACK. This is because HACK is able to leverage upon the out of order packet retransmission algorithms in SACK. HACK creates these out of order situations as it may not be able to recover the headers of all the packets corrupted in a burst due to the random nature of the bit errors within each packet. For example, if say 5 packets are corrupted, HACK may only be able to recover the headers of packets 2, 4 and 5 with packets 1 and 3 being irretrievable. This creates gaps in the receiving window, as HACK will only ask for retransmissions of the packets whose headers it can recover. However, if SACK is activated, these gaps will be detected and handled accordingly. Another example would be as follows; suppose the TCP receiver receives segments x+1, x+2 and x+3 correctly but segment x is corrupted. In this case, the receiver will generate one 'special' ACK in response to segment x and three normal ACKs in response to segments x+1, x+2 and x+3. However, the three normal ACKs will appear to the TCP sender as dupacks as they all will be acknowledging segment x (the next segment expected by the receiver). Hence, the sender will needlessly go into fast retransmit. SACK eliminates this problem as it will be able to inform the TCP sender about the gaps in the receiving window. This leveraging is possible because the HACK and SACK have disjoint sets of operations, thus preventing any conflicts during packet processing. However, it must be re-emphasised that HACK without SACK is still much better than just SACK alone (albeit with potentially more out of order packets being generated). Thus both SACK and HACK can benefit very nicely from each other's properties.

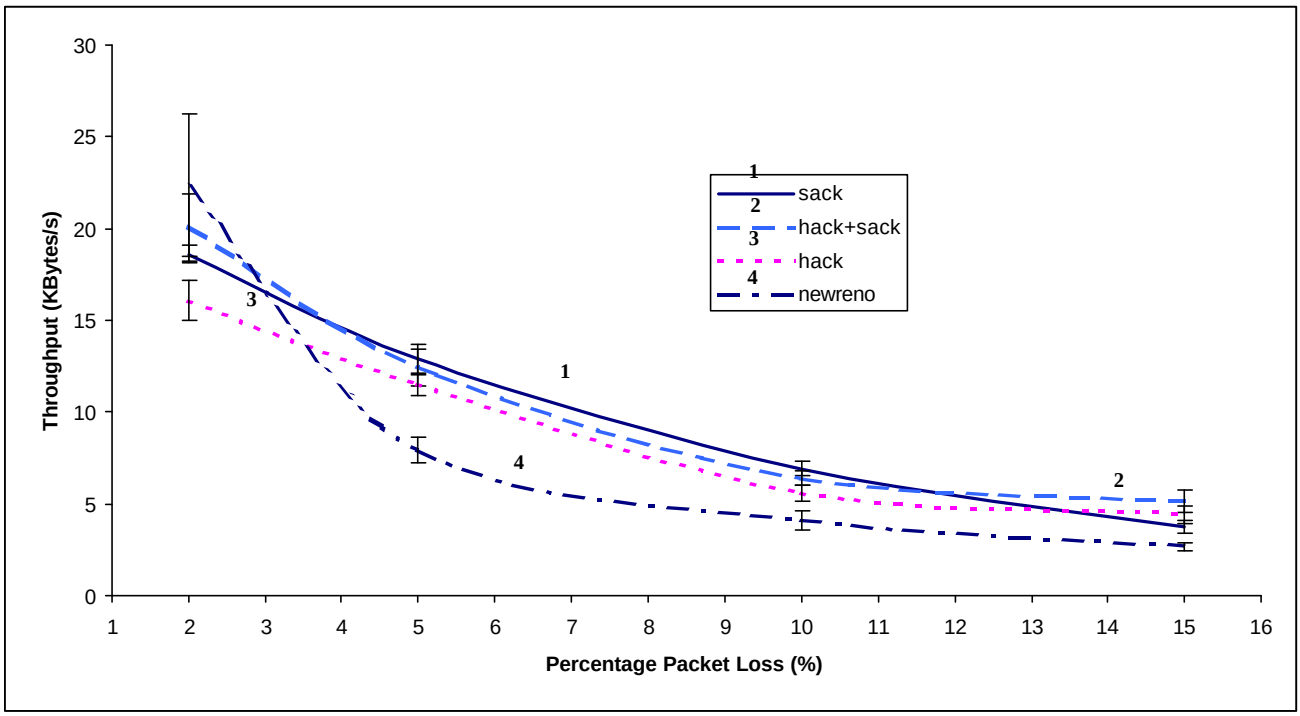

Fig 9. Throughput versus percentage packet loss for long latency (300 ms) link with random single packet errors

To clearly show how well HACK performs in bursty error conditions, we compared the time sequence graphs (TSG) of HACK, SACK and HACK+SACK for 5% error probability with a burst error length of 5 packets. Tcptrace [Ost97] was used to generate the TSG graphs (as xplot [She97] data files) from our tcpdump capture files of the data transferred between the server and the client during the experiment. Xplot was used to display the TSG graphs and we captured the output on the screen using a screen capture utility. The TSG for HACK+SACK is shown in Fig. 14, HACK in Fig. 15 and SACK in Fig. 16. It can be seen that HACK and HACK+SACK perform much better than SACK in keeping the data pipe flowing in the presence of burst errors as they do not have long periods of idle activity / timeouts (shown as long

horizontal lines in the TSG indicating that the sequence number for the TCP connection has not increased during that time period). HACK+SACK works better than HACK due to the reasons mentioned previously. It must be noted that the time

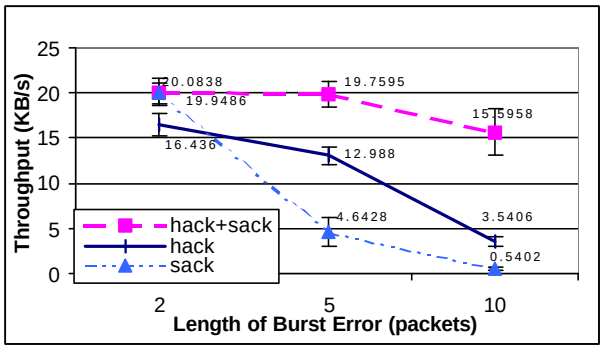

Fig 10. Throughput for 2% burst error for various burst lengths

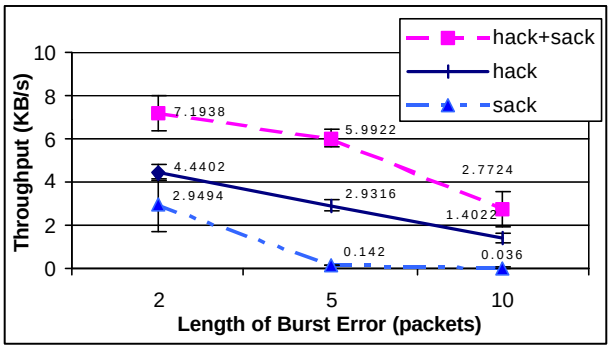

Fig 12. Throughput for 10% burst error for various burst lengths

scale of the various TSG graphs are different and that SACK takes a much longer time to finish than both HACK and HACK+SACK. This is shown in Fig. 17 which displays the instantaneous throughput versus time. As can be seen, SACK takes about 2600 seconds to finish as compared to about 430 seconds for HACK and 140 seconds for HACK+SACK. Thus HACK and HACK+SACK enjoy a much higher throughput than SACK in bursty error conditions.

5.3 Effect of Window Sizes

So far in our experiments, we kept the window size large enough not to be a bottleneck. Next, we consider the effect of smaller window size on the performance of HACK and SACK. It is clear that when there are a number of errors and window size is small, more timeouts and hence slow starts are likely to occur, resulting in throughput degradation. However, HACK will keep the pipe flowing because of the `special' ACKs, and hence will result in better throughput. To confirm this, we compared the effects of various window sizes on SACK and HACK. We ran this experiment over long latency links for burst errors with burst lengths ranging from 1 to 10 packets. We transferred 256 KB of data from the client to the server with two distinct window sizes: 16KB and 64KB. Figs. 18 – 20 show

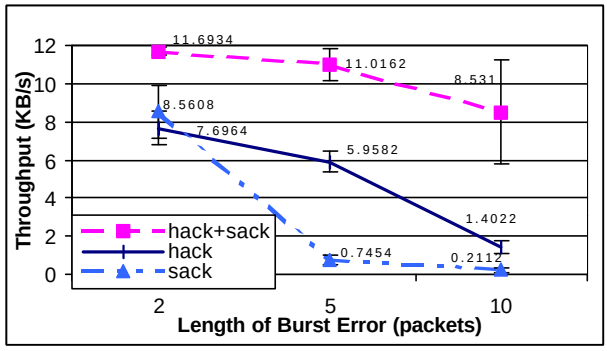

Fig 11. Throughput for 5% burst error for various burst lengths

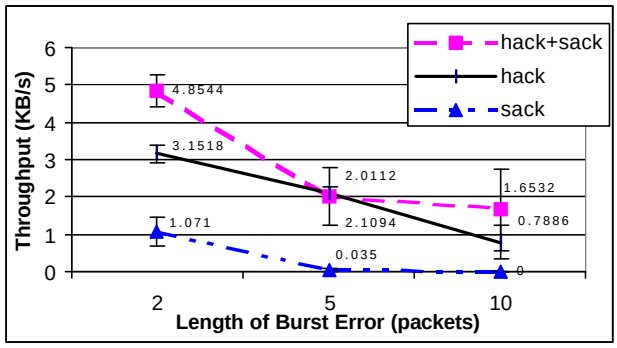

Fig 13. Throughput for 15% burst error for various burst lengths

the throughput of the various TCP schemes under different burst error lengths for burst error probabilities of 2%, 5%, and 10%, respectively, for a window size of 16 KB. As can be seen, HACK performs better than SACK even when the window size is small (thus becoming a limiting factor in determining the amount of data that can be sent over a link), and HACK+SACK performs better than HACK. The reasons for these improvement are as stated previously. Figs. 21 - 23 show the results for the same error probabilities and burst lengths but for a window size of 64 KB. In this case as well, HACK performs much better than SACK and HACK+SACK performs better than HACK.

These results clearly show that HACK performs better than SACK in bursty error conditions for window sizes which are typically used by many TCP stacks (without the optional window scaling option enabled). Note that the superior performance of HACK over SACK is more prominent for smaller window size.

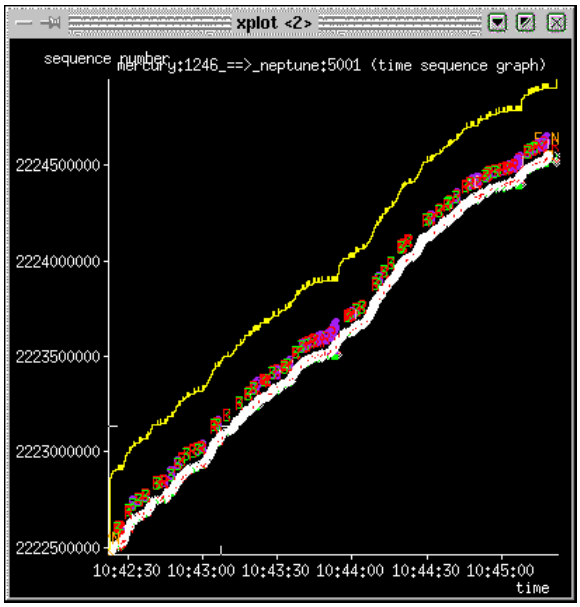

Fig 14. Time Sequence Graph for HACK+SACK

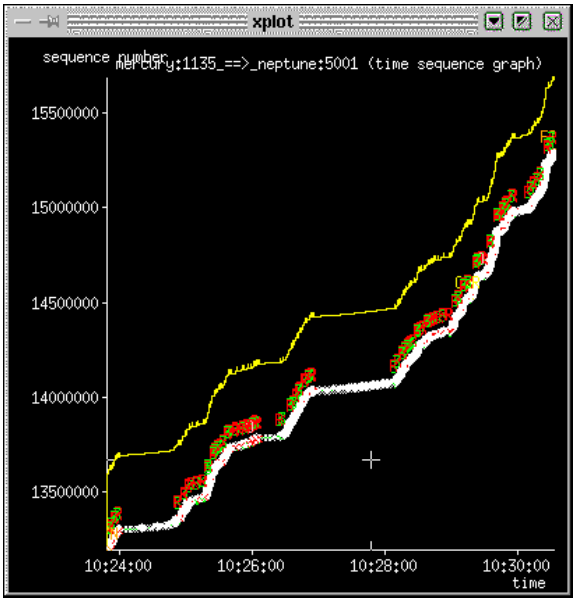

Fig 15. Time Sequence Graph for HACK

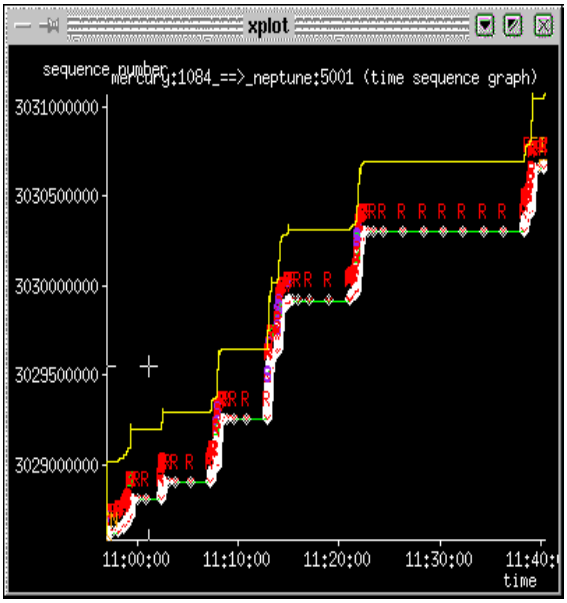

Fig 16. Time Sequence Graph for SACK

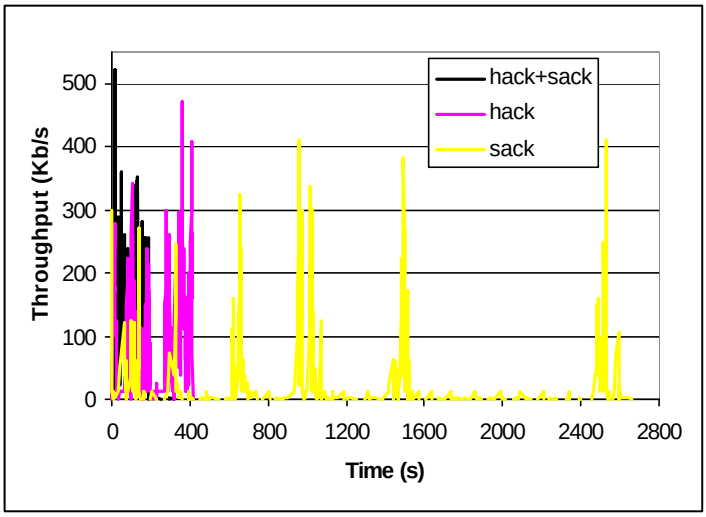

Fig 17. Throughput versus Time graph for various TCP implementations

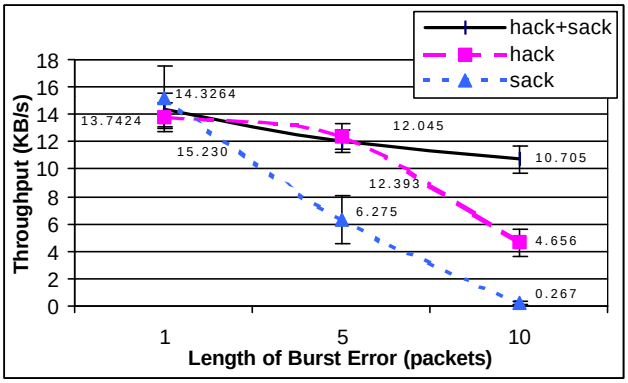

Fig 18. Throughput for 2% burst error for various burst lengths (window size of 16KB)

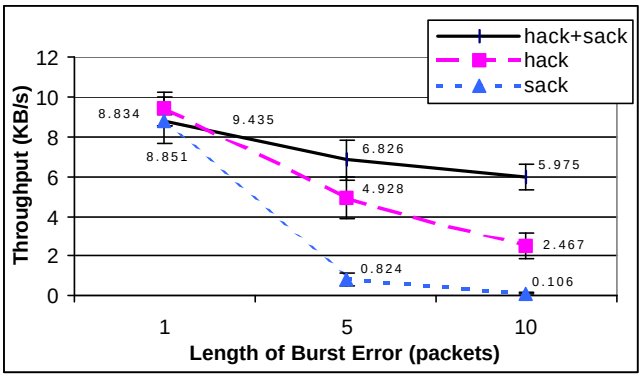

Fig 19. Throughput for 5% burst error for various burst lengths (window size of 16KB)

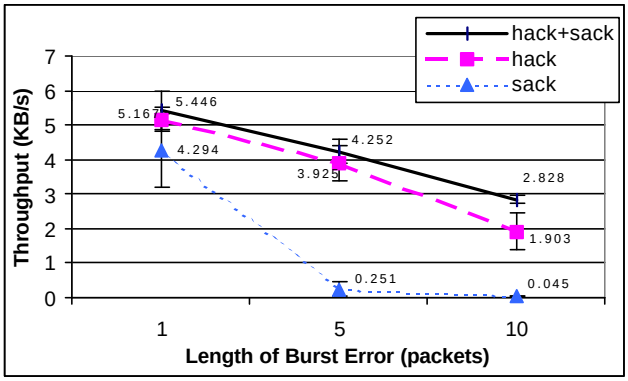

Fig 20. Throughput for 10% burst error for various burst lengths (window size of 16KB)

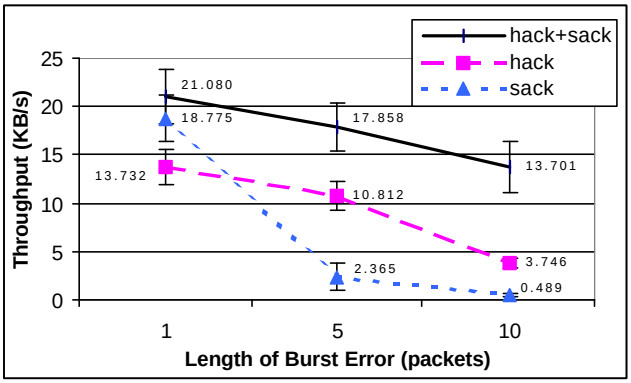

Fig 21. Throughput for 2% burst error for various burst lengths (window size of 64KB)

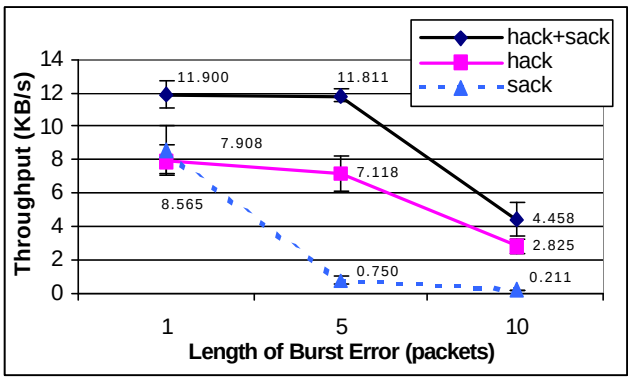

Fig 22. Throughput for 5% burst error for various burst lengths (window size of 64KB)

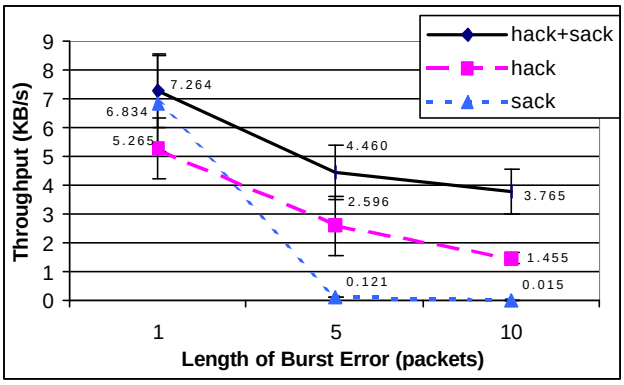

Fig 23. Throughput for 10% burst error for various burst lengths (window size of 64KB)

6 Deployment

In scenarios where it may be difficult to determine if HACK is necessary (e.g., if the end user is unaware of the existence of any lossy links within the network), a feasible solution would be to place TCP tunnels (similar to IP tunnels except that TCP is used for the encapsulation) across those links and enable HACK for those tunnels.

These tunnels would be deployed by the network administrators of the lossy links. Traffic entering these lossy links will be encapsulated within TCP tunnels and these tunnels can then use the Header Checksum option to maximise their throughput over these lossy links. In this scenario, the end users do not have to change any of their software or even be aware of the presence of lossy links in the network to benefit from the use of the Header Checksum option. The properties of TCP tunnels is described in [LBJSA] and a complete system which provides quality of service (QoS) guarantees while using TCP tunnels is described in [BLJSA].

7 Conclusions

In this paper, we have presented the TCP Header Checksum extensions to TCP to recover from packet loss due to corruption in lossy environments. HACK allows TCP to detect packet loss due to corruption and recover the necessary information so that the sender may be notified of this corruption allowing it to retransmit the corrupted segment immediately. The sender avoids throttling its sending rate as the loss is not indicative of congestion.

Our experiments have shown that HACK performs substantially better than SACK in environments where burst corruptions are prevalent. In these environments, SACK will time out incessantly whereas HACK manages to keep the data pipe flowing somewhat. The optimal level of performance is achieved when HACK is run together with SACK.

Work is being done to test the effectiveness of HACK and HACK + SACK in situations where ACKs are also susceptible to packet corruption, and

where congestion occurs along with corruption. We also plan to extend our test and measurements of HACK to real wireless and satellite links.

References

[BLJSA] R. K. Balan, B. P. Lee, L. Jacob, W. K. G. Seah, A. L. Ananda, "Adaptive QoS Provisioning for Traffic between Border Routers", Submitted to ICNP 2000.

[BB] A. Bakre, B. R. Badrinath, "Hand-off and System Support for Indirect TCP/IP", Proceedings of Second Usenix Symposium on Mobile and Location-Independent Computing, April 1995.

[BC98] P. Barford and M. Crovella, "Generating Representative Web Workloads for Network and Server Performance Evaluation", Proc. ACM SIGMETRICS '98, June 1998

[BHZ] R. Bruyeron, B. Hemon, L. Zhang, "Experimentations with TCP Selective Acknowledgement", ACM SIGCOMM

[BKVP] B. S. Bakshi, P. Krishna, N. H. Vaidya, D. K. Pradhan, "Improving Performance of TCP over Lossy Networks", Texas A&M University.

[BSAK] H. Balakrishnan, S. Seshan, E. Amir, R. H. Katz, "Improving TCP/IP Performance over Lossy Networks", Proc. 1st ACM Int'l Conf. On Mobile Computing and Networking (Mobicom), Nov 1995.

[BPSK] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, R. H. Katz, "A Comparison of Mechanisms for Improving TCP Performance over Lossy Links", IEEE/ACM Transactions on Networking, Dec 1997.

[FF96] K. Fall, S. Floyd, "Simulation-based Comparisons of Tahoe, Reno, and SACK TCP", Computer Communications Review, July 1996.

[Flo94] S. Floyd, "TCP and Explicit Congestion Notification", ACM CCR, October 1994.

[Hoe95] J. Hoe, "Startup Dynamics of TCP's Congestion Control and Avoidance Schemes", Master's Thesis, MIT, 1995

[Hoe96] J. Hoe, "Improving the Startup Behaviour of a Congestion Control Scheme for TCP", ACM SIGCOMM 1996.

[Jac88] Van Jacobson, "Congestion Avoidance and Control", ACM SIGCOMM 1988.

[LBJSA] B. P. Lee, R. K. Balan, L. Jacob, W. K. G. Seah, A. L. Ananda, "TCP Tunnels: Avoiding Congestion Collapse", Submitted to LCN 2000.

[MM96] M. Mathis, J. Mahdavi, "Forward Acknowledgement: Refining TCP Congestion Control", ACM SIGCOMM 1996.

[Ost97] S. Ostermann, "tcptrace", http://jarok.cs.ohiou.edu/software/tcptrace/tcptrace.html, Dec 1997

[RFC793] J. Postel, "Transmission Control Protocol", RFC793

[RFC2488] M. Allman, D. Glover, L. Sanchez, "Enhancing TCP Over Satellite Channels using Standard Mechanisms", RFC2488

[RFC2581] M. Allman, V. Paxson, W. Stevens, "TCP Congestion Control", RFC2581

[RFC2018] M. Mathis, J. Mahdavi,S. Floyd, A. Romanow , "TCP Selective Acknowledgment Options", RFC2018

[RFC2481] K. K. Ramakrishnan, S. Floyd, “A Proposal to add Explicit Congestion Notification (ECN) to IP”, RFC2481

[RFC2582] S. Floyd, T. Henderson, “The NewReno Modification to TCP's Fast Recovery Algorithm”, RFC2481

[She97] T. Sheppard, “xplot”, ftp://mercury.lcs.mit.edu/pub/shep, Aug 1997

[S94] W. R. Stevens, "TCP/IP Illustrated, Volume 1", Addison-Wesley Publishing Company, 1994.