ELSEVIER

Computer Networks 39 (2002) 347–361

COMPUTER NETWORKS

www.elsevier.com/locate/comnet

TCP HACK: a mechanism to improve performance over lossy links ☆

R.K. Balan [1], B.P. Lee [2], K.R.R. Kumar, L. Jacob, W.K.G. Seah, A.L. Ananda *

Department of Computer Science, Centre for Internet Research, School of Computing, National University of Singapore, 3 Science Drive 2, Lower Kent Ridge Road, Singapore 117543, Singapore

Received 12 October 2001; accepted 18 December 2001

Responsible Editor: I.F. Akyildiz

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 selective acknowledgement (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. © 2002 Elsevier Science B.V. All rights reserved.

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.

☆ A version of this paper was presented in the IEEE INFOCOM 2001 conference, Anchorage, Alaska.

* Corresponding author.

E-mail addresses: rajesh@cs.cmu.edu (R.K. Balan), lee.bp@unitywireless.com.sg (B.P. Lee), kaleelaz@comp.nus.edu.sg (K.R.R. Kumar), jacobl@comp.nus.edu.sg (L. Jacob), wseah@comp.nus.edu.sg (W.K.G. Seah), ananda@comp.nus.edu.sg (A.L. Ananda).

[1] Present address: Carnegie Mellon University, School of Computer Science, 5000 Forbes Avenue, Pittsburgh, PA 15232, USA.

[2] Present address: Unity Wireless Integration, 1123 Serangoon Road, #02-01 UMW Building, Singapore 328207, Singapore.

1389-1286/02/$ - see front matter © 2002 Elsevier Science B.V. All rights reserved.
PII: S1389-1286(01)00310-3

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 [12]. 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 [17,19,24] 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.

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 [19] 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 [10,11,22] 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 [20] 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. Allman et al. [18] covers the issues in using FEC to improve the performance of satellite links. The indirect-TCP (I-TCP) protocol [2] 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 [6] caches packets at the base station and performs local retransmissions over the lossy link.

The use of explicit congestion notification (ECN) [9,21] 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) [7], explicit bad state notification (EBSN) [5] and forward acknowledgement (FACK) [15].

The use of a checksum to protect header information has also been proposed by Larzon et al. [13] in their UDP lite protocol. They allow applications to receive UDP data even if the data had been corrupted as their claim is that many UDP applications would prefer to receive slightly corrupted data instead of no data.

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 (HACK) 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 of the Header Checksum option. The rest of the data sending algorithm is as per normal (Fig. 3).

3.2. Modifications to the TCP receiver

When the TCP receiver (Fig. 4) 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.

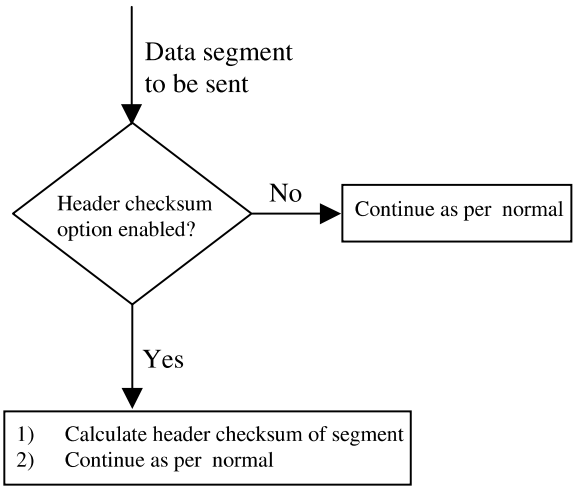

Fig. 3. Modifications to the TCP sender.

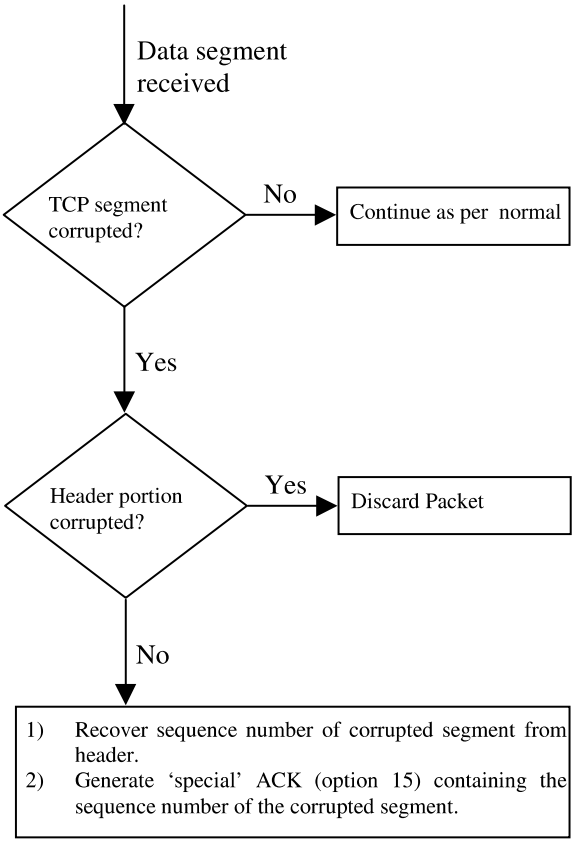

Fig. 4. Modifications to the TCP receiver.

(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.

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:

(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, Fig. 5).

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 MB of RAM each. The machines were connected using Intel Ether-Express

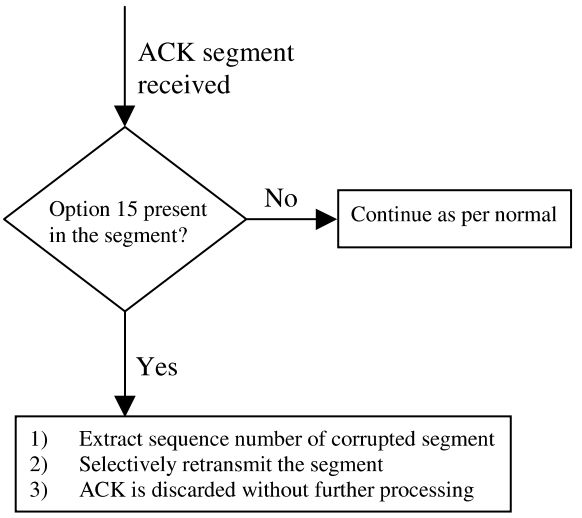

Fig. 5. Modification to the ACK processing.

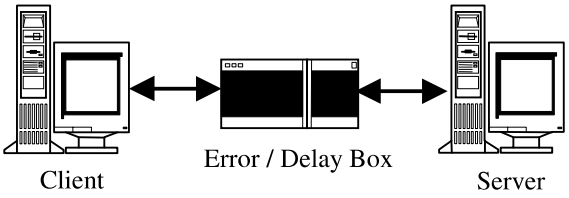

Fig. 6. Experimental testbed.

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* [3] 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, 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. Infact, layer-2 protocol delivering corrupt PDUs to upper layer can happen in the following scenarios (i) the user chooses the option of disabling the link-layer CRC and (ii) the error rate is too high for the link-layer CRC to be efficient. We claim that disabling error detection and recovery at lower layers and letting TCP to do them can be beneficial to the overall performance.

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 [4,8,22].

We used packet corruption probabilities ranging from 2% to 15%, which corresponds to a bit error rate of 1×10^{-6}–1×10^{-5}.

For each of the packet corruption probabilities given above, we still had to determine what proportion of the packet headers were corrupted. This metric was fundamental in determining HACKs effectiveness, as HACK requires uncorrupted packet headers. Since the actual value of this metric is heavily dependent on the environment, we elected to use two different values that reflect the entire range of values that this metric could possibly take. In the first case, we assumed that 0% of the headers in corrupted packets were corrupted, i.e., this would be the best possible case for HACK. In the second case, we assumed that 95% of the packet headers were corrupted. This would represent almost the worst possible case for HACK. We are confident that the actual header corruption percentage in any wireless environment would fall somewhere between these two extremes.

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 characterised by sharp spikes causing single bit corruptions. We have translated this bit error profile into packet errors (as mentioned earlier, our packet corruption probabilities range from 2% to 15% corresponding to a bit error range of 1×10^{-6}–1×10^{-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 latter case, we used 256 KB 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. We also repeated each experiment using the two different header corruption probabilities (0% and 95% as explained above). The TCP window size was set high enough that it was not a limiting factor for either latency. Average throughput versus percentage loss curves are shown in Fig. 7 for the low latency link with 95% header corruption and in Fig 8 for 0% header

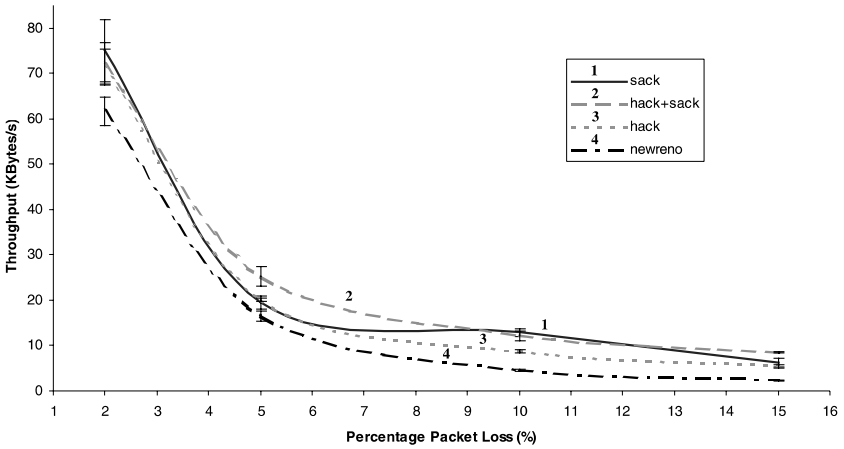

Fig. 7. Throughput versus percentage packet loss for low latency (10 ms) link with random single packet error (95% header error).

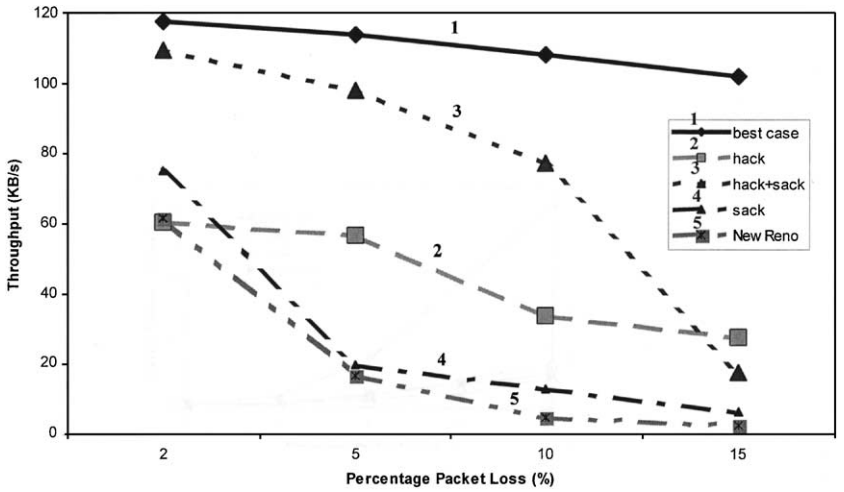

Fig. 8. Throughput versus percentage packet loss for low latency (10 ms) link with random single packet errors (0% header error).

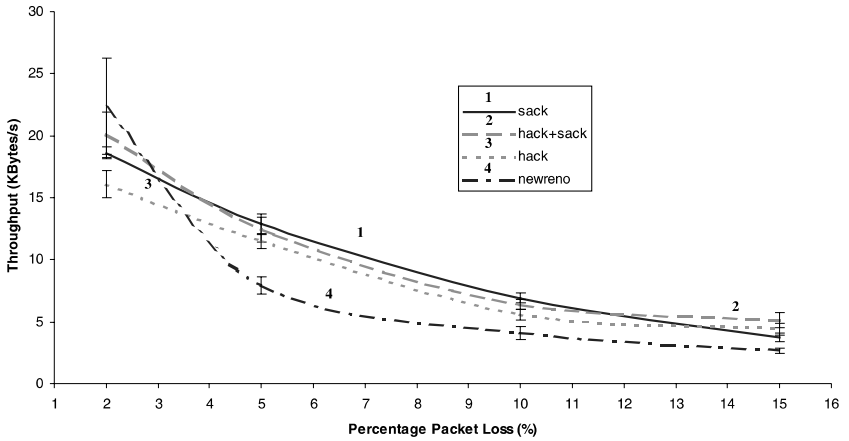

Fig. 9. Throughput versus percentage packet loss for long latency (300 ms) link with random single packet errors (95% header error).

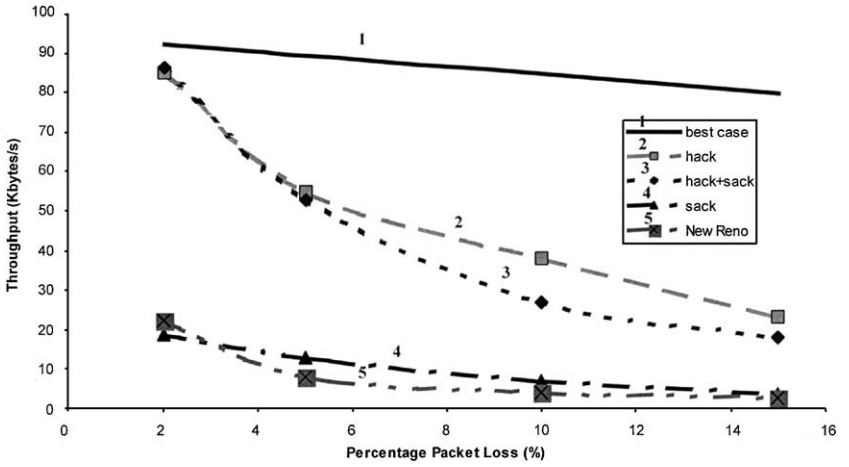

Fig. 10. Throughput versus percentage packet loss for long latency (300 ms) link with random single packet errors (0% header error).

corruption. Fig. 9 shows the results for the long latency link with 95% header corruption and Fig. 10 shows the results for the long latency link with 0% header corruption. Fig. 11 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 and for both header corruption percentages. 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. The results show that HACK still works well even when it can only recover the header information from a small percentage (5%) of the corrupted packets. Based on these results, 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.

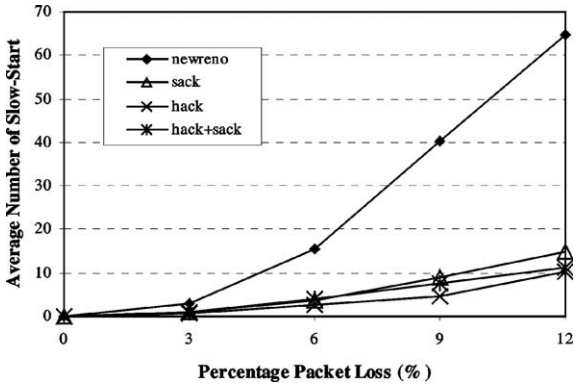

Fig. 11. Average number of slow starts for long latency link with random single packed errors (95% header error).

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. Figs. 12–15 show the results of the various TCP schemes under different burst error lengths for 2%, 5%, 10% and 15% burst error probabilities respectively with a header corruption percentage of 0%. Figs. 16–19 show the same results but for a header corruption percentage of 95%.

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

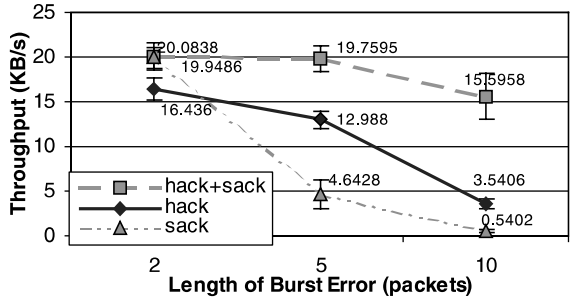

Fig. 12. Throughput for 2% burst error for various burst lengths (95% header corruption).

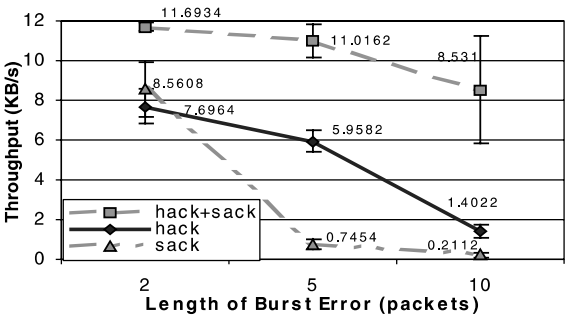

Fig. 13. Throughput for 5% burst error for various burst lengths (95% header corruption).

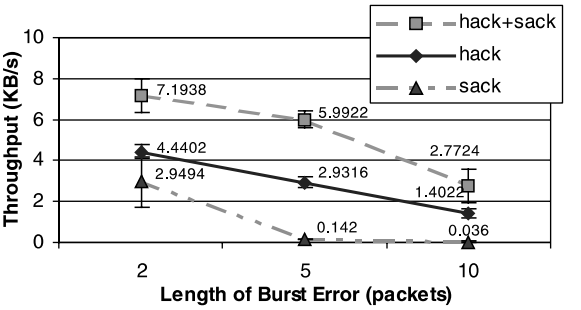

Fig. 14. Throughput for 10% burst error for various burst lengths (95% header corruption).

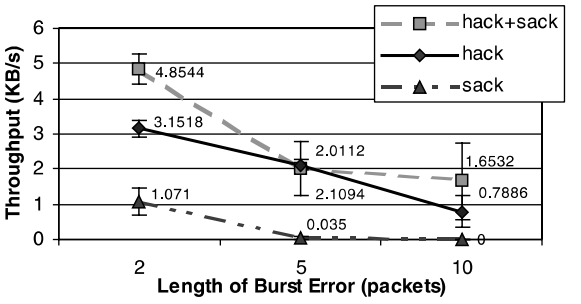

Fig. 15. Throughput for 15% burst error for various burst lengths (95% header corruption).

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

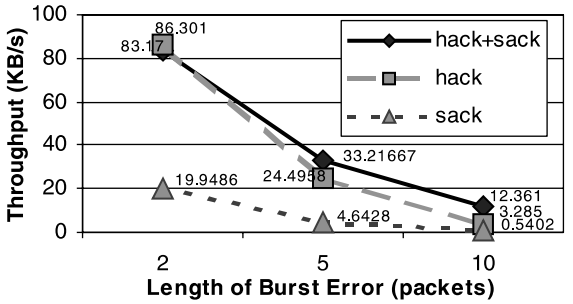

Fig. 16. Throughput for 2% burst error for various burst lengths (0% header corruption).

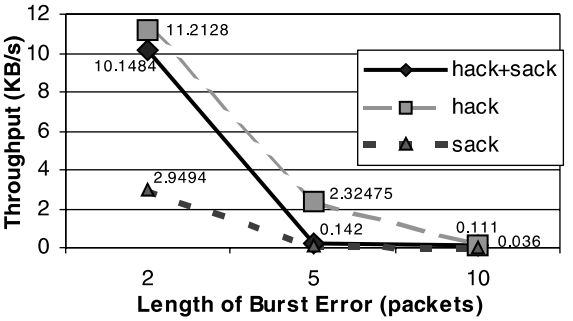

Fig. 17. Throughput for 5% burst error for various burst lengths (0% header corruption).

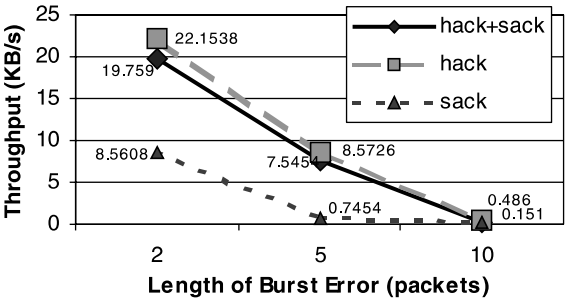

Fig. 18. Throughput for 10% burst error for various burst lengths (0% header corruption).

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 five packets are corrupted, HACK

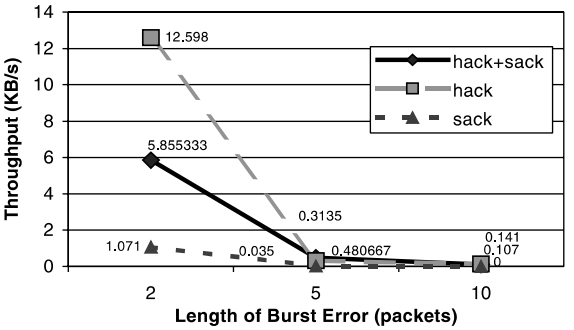

Fig. 19. Throughput for 15% burst error for various burst lengths (0% header corruption).

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.

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 five packets and a header corruption percentage of 95%. Tcptrace [16] was

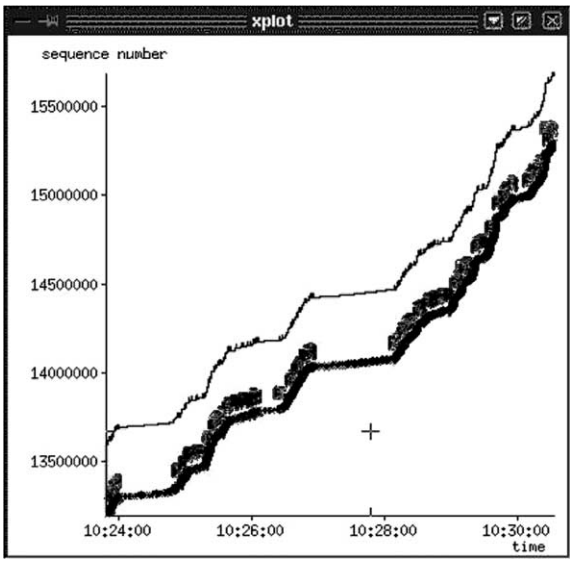

Fig. 20. TSG for HACK (95% header corruption).

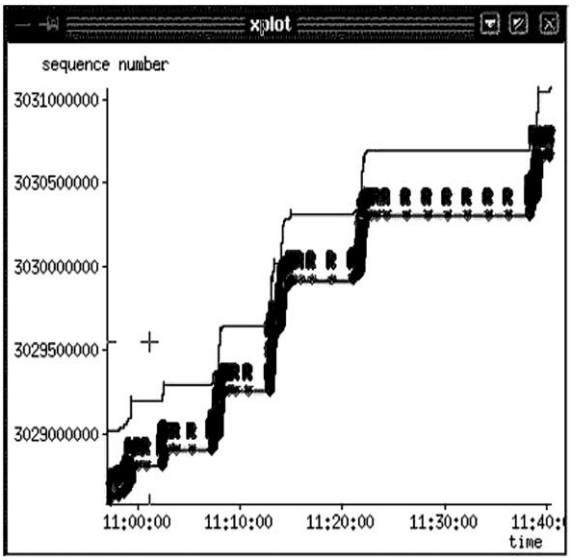

Fig. 21. TSG for SACK (95% header corruption).

used to generate the TSG graphs (as *x*plot [23] data files) from our tcpdump capture files of the data transferred between the server and the client during the experiment. *x*plot was used to display the TSG graphs and we captured the output on the screen using a screen capture utility. The TSG for HACK is shown in Fig. 20, SACK in Fig. 21 and HACK + SACK in Fig. 22. 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 scale of the various TSG graphs are different and that SACK takes a much longer time to finish than HACK and HACK + SACK as shown in Fig. 23 which displays the instantaneous throughput versus time.

Fig. 24 shows this result in the form of a bar chart. As can be seen, SACK takes about 2600 s to finish as compared to about 430 s for HACK and 140 s for HACK + SACK. Thus HACK and HACK + SACK enjoy a much higher throughput

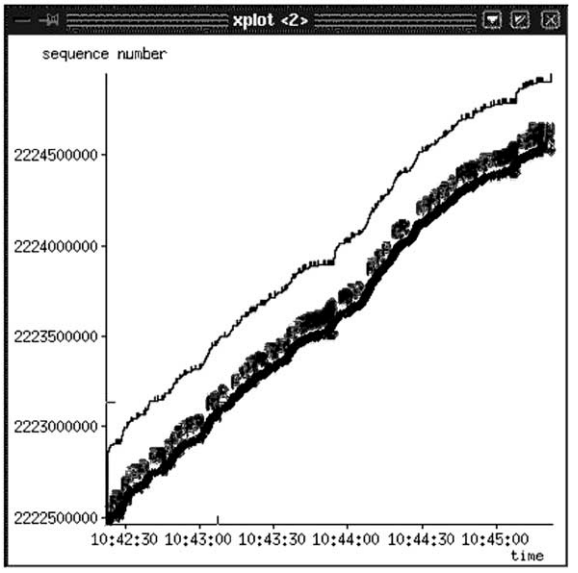

Fig. 22. TSG for HACK + SACK (95% header corruption).

than SACK in bursty error conditions even in situations where HACK is only able to recover the headers from a small percentage of the corrupted packets. Fig. 25 shows the time taken for the three algorithms to finish the same experiment but with a header corruption percentage of 0%. As can be

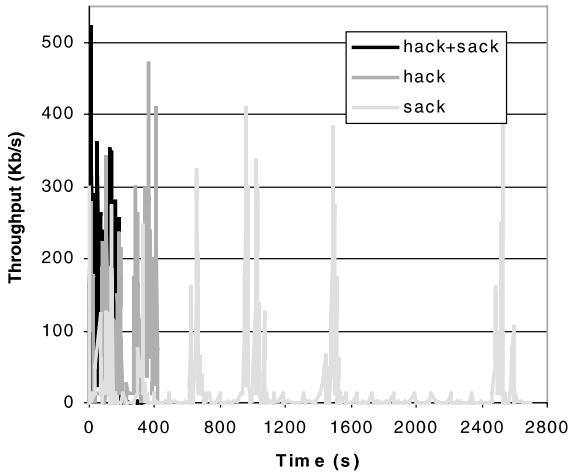

Fig. 23. Throughput versus time graph for various TCP implementations (95% header corruption).

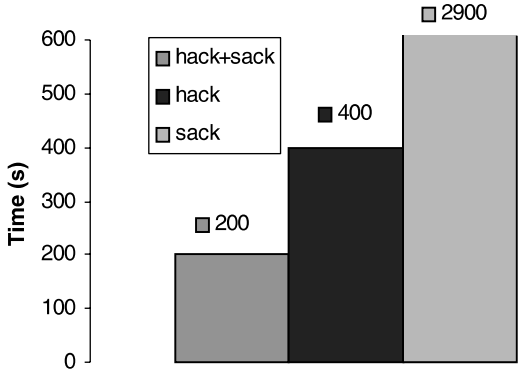

Fig. 24. Throughput versus time graph for various TCP implementations 5% burst error with 95% header corruption.

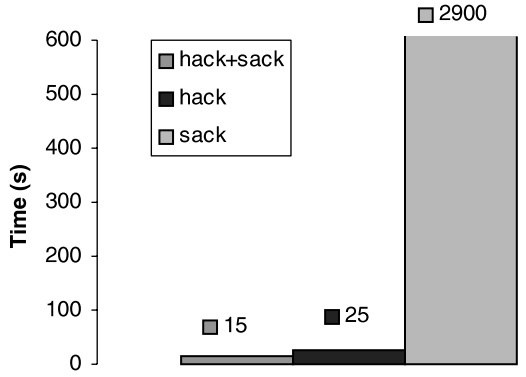

Fig. 25. Throughput versus time graph for various TCP implementations 5% burst error with 0% header corruption.

Table 1
Summary of results

Header corruption		
Error types	0%	95%
Random errors (long and short latencies)	5–10× better than SACK	Equal to SACK
Burst errors (long latency)	100× better than SACK	6× better than SACK

seen, HACK is much better when it is able to recover the headers of all corrupted packets.

Table 1 summarises the results for random bit errors and burst errors. As can be seen, HACK performs better than SACK in most of the cases. This improvement of performance is seen even when 95% of the headers of corrupted packets by HACK (which represents almost the worst are unrecoverable possible case for HACK).

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 and only for a header corruption percentage of 95%. We transferred 256 KB of data from the client to the server with two distinct window sizes: 16 and 64 KB. Figs. 26–28 show 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

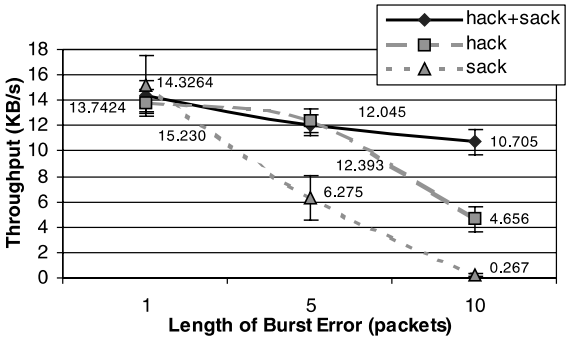

Fig. 26. Throughput for 2% burst error for various burst lengths (window size of 16 KB, 95% header corruption).

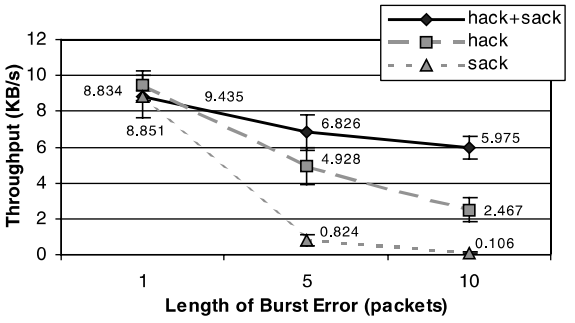

Fig. 27. Throughput for 5% burst error for various burst lengths (window size of 16 KB, 95% header corruption).

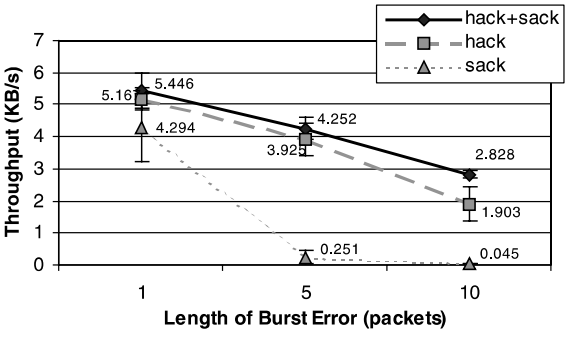

Fig. 28. Throughput for 10% burst error for various burst lengths (window size of 16 KB, 95% header corruption).

than HACK. The reasons for these improvements are same as stated previously. Figs. 29–31 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.

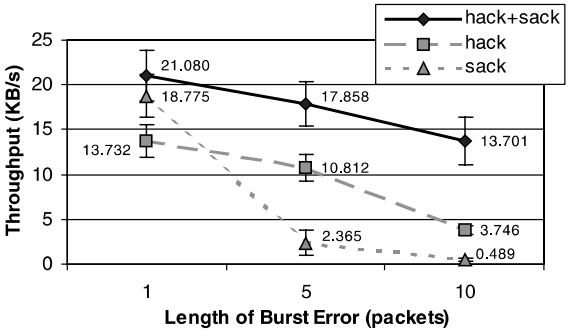

Fig. 29. Throughput for 2% burst error for various burst lengths (window size of 64 KB, 95% header corruption).

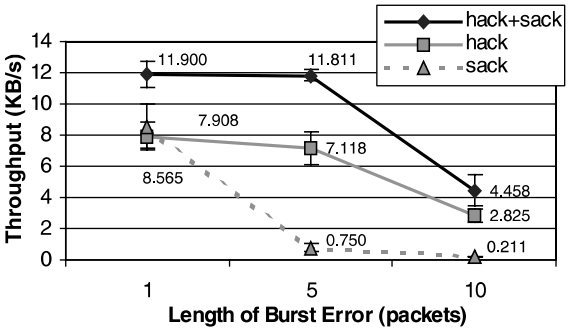

Fig. 30. Throughput for 5% burst error for various burst lengths (window size of 64 KB, 95% header corruption).

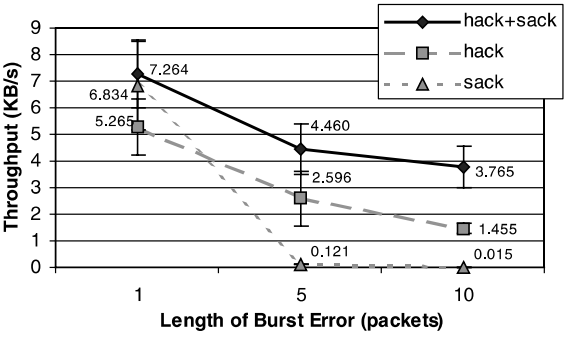

Fig. 31. Throughput for 10% burst error for various burst lengths (window size of 64 KB, 95% header corruption).

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.

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 [14] and a complete system which provides quality of service (QoS) guarantees while using TCP tunnels is described in [1].

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 timeout incessantly whereas HACK manages to keep the data pipe flowing somewhat. HACK manages to provide 6× better throughput than SACK in the presence of burst errors even when 95% of the headers of corrupted packets are unrecoverable. In the case where HACK can recover the headers of all corrupted packets, it outperforms SACK by about 100× in the presence of burst errors. 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.

Acknowledgements

The authors would like to acknowledge the National Science and Technology Board (NSTB) of Singapore and the Singapore Advanced Research and Education Network (SingAREN) project for supporting the above work under the Broadband21 project grant.

References

[1] R.K. Balan, Chameleon—a system for adaptive QoS provisioning, Master's Thesis, School of Computing, NUS, 2000.
[2] A. Bakre, B.R. Badrinath, Handoff and system support for indirect TCP/IP, Proceedings of Second Usenix Symposium on Mobile and Location-Independent Computing, April 1995.
[3] University of Illinois at Urbana Champagne, available from http://dast.nlanr.net/Projects/Iperf.
[4] R. Bruyeron, B. Hemon, L. Zhang, Experimentations with TCP selective acknowledgement, Proceedings of ACM SIGCOMM, April 1998.
[5] B.S. Bakshi, P. Krishna, N.H. Vaidya, D.K. Pradhan, Improving performance of TCP over wireless networks, Proceedings of 17th IEEE Internation Conference on Distributed Computer Systems (ICDSC), 1997.
[6] H. Balakrishnan, S. Seshan, E. Amir, R.H. Katz, Improving TCP/IP performance over lossy networks, Proceedings of 1st ACM International Conference On Mobile Computing and Networking (Mobicom), November 1995.
[7] 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, December 1997.
[8] K. Fall, S. Floyd, Simulation-based comparisons of Tahoe, Reno, and SACK TCP, ACM Computer Communications Review 26 (3) (1996) 5–21.
[9] S. Floyd, TCP and explicit congestion notification, ACM Computer Communications Review 24 (5) (1994) 10–23.
[10] J. Hoe, Startup dynamics of TCP's congestion control and avoidance schemes, Master's thesis, MIT, 1995.

[11] J. Hoe, Improving the startup behaviour of a congestion control scheme for TCP, Proceedings of ACM SIGCOMM, 1996.
[12] Van Jacobson, Congestion avoidance and control, Proceedings of ACM SIGCOMM, 1988.
[13] L.-Å. Larzon, M. Degermark, S. Pink, UDP lite for real-time multimedia applications, Proceedings of the IEEE International Computer Communications (ICC'99) Conference, June 1999.
[14] B.P. Lee, R.K. Balan, L. Jacob, W.K.G. Seah, A.L. Ananda, TCP tunnels: avoiding congestion collapse, Proceedings of 25th Annual IEEE Conference on Local Computer Networks (LCN), November 2000.
[15] M. Mathis, J. Mahdavi, Forward acknowledgement: refining TCP congestion control, ACM SIGCOMM, 1996.
[16] S. Ostermann, tcptrace, available from http://jarok.cs.ohiou.edu/software/tcptrace/tcptrace.html, December 1997.
[17] J. Postel, Transmission control protocol, RFC793.
[18] M. Allman, D. Glover, L. Sanchez, Enhancing TCP over satellite channels using standard mechanisms, RFC2488.
[19] M. Allman, V. Paxson, W. Stevens, TCP congestion control, RFC2581.
[20] M. Mathis, J. Mahdavi, S. Floyd, A. Romanow, TCP selective acknowledgment options, RFC2018.
[21] K.K. Ramakrishnan, S. Floyd, A proposal to add explicit congestion notification (ECN) to IP, RFC2481.
[22] S. Floyd, T. Henderson, The NewReno modification to TCP's fast recovery algorithm, RFC2582.
[23] T. Sheppard, xplot, available from ftp://mercury.lcs.mit.edu/pub/shep, August 1997.
[24] W.R. Stevens, in: TCP/IP Illustrated, Vol. 1, Addison-Wesley, Reading, MA, 1994.

Rajesh Krishna Balan obtained his Bachelors in Computer Science (with honours) and Masters in Computer Science from the National University of Singapore in 1998 and 2001 respectively. Presently he is pursuing a Ph.D. in Computer Science at Carnegie Mellon University. He is currently working in the area of ubiquitious computing from the Operating System point of view.

Lee Boon Peng completed his MSc (Computer Science) at the School of Computing, National University of Singapore. He is currently an engineer with Unity Integration Corp., a total solutions provider in Transportation and Mobile Information Management Systems, while maintaining an active research interest in network protocol design and performance.

K.R. Renjish Kumar (renjish@ieee.org) received the Bachelor of Engineering degree in Electronics and Communications from the Regional Engineering College, Suratkal, India in 1997. He is currently pursuing Masters in Computer Science in the area of quality of services in networks at the Centre for Internet Research, National University of Singapore. He was with Cognizant Technology Solutions for over two years and is currently working with Siemens ICM, Singapore as R&D Engineer. His research interests are IP QoS, TCP performance issues, wireless networks, mobile communication.

Lillykutty Jacob obtained her M. Tech. degree in electrical engineering (communication engineering) from the Indian Institute of Technology at Madras in 1985, and PhD degree in electrical communication engineering (computer networks) from the Indian Institute of Science, Bangalore, in 1993. She was a research fellow in the department of computer science, Korea Advanced Institute of Science and Technology, S. Korea, during 1996-'97. Since 1985 she has been with the Regional Engineering College at Calicut, India. Currently, she is with the School of Computing, National University of Singapore, where she is a visiting academic fellow. Her research interests include Quality-of-Service and Resource Management in Internet, Network Protocols, and Performance Modelling and Analysis. She is a member of IEEE.

Dr Winston Seah is the Programme Director of the Internet Technologies programme in CWC. Prior to joining CWC, he worked in the Department of Electrical Engineering, National University of Singapore (NUS), and the Ministry of Defence.

Dr Seah received the Dr.Eng. degree from Kyoto University, Kyoto, Japan, in 1997 and, the M.Eng. (Electrical Engineering) and B.Sc. (Computer and Information Sciences) degrees from NUS in 1993 and 1987 respectively. For his postgraduate studies, he received the Monbusho Postgraduate Scholarship from the Government of Japan, as well as scholarships from the Foundation for C&C Promotion (NEC funded) and the International Communication Foundation (KDD funded) in Japan.

Dr Seah also holds joint teaching positions in the Department of Electrical and Computer Engineering, and Department of Computer Science in NUS, where he lectures in mobile computing and computer networks courses. He is actively involved in research and development in the areas of mobile/wireless Internet technologies, mobile ad hoc networks and Internet quality of service (QoS).

Akkihebbal L. Ananda is an Associate Professor in the Computer Science Department of the School of Computing at the National University of Singapore. He is also the Director of the Centre for Internet Research. He is actively associated with Singapore Advanced Research and Education Project (SingAREN) and has involved in network research and connectivity issues relating to Internet2. He is one of the key players in developing the NUS's campus secure plug-and-play network which has around 12,000 points campus wide. His research areas of interest include High-speed computer networks, transport protocols, collaborative applications, and distributed systems. He is a member of the IEEE Computer and Communications Societies.

Ananda obtained his B.E. degree in electronics from the University of Bangalore, India, in 1971; his M.Tech degree in Electrical Engineering from the Indian Institute of Technology, Kanpur in 1973, and the M.Sc and Ph.D degrees in computer science from the University of Manchester, UK, in 1981 and 1983 respectively. From 1974 to 1980 he worked as a system software engineer in India.