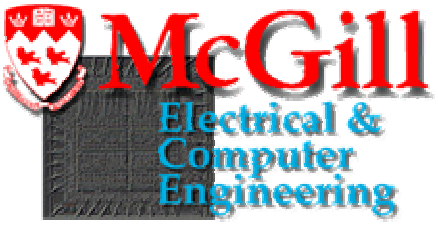
Experiment 4
TCP

Submitted by: Huang, Jim-Chet and Khazzam, Shawn
Student ID: 110034835, 119921328



Course Number: ECSE-489B
Date: March 21st, 2003

1 Abstract

TCP Reno, in its current implementation, suffers in performance when multiple packets are dropped per transmit window. Also, TCP Reno's congestion window sizes are presently constrained by its congestion control mechanisms. Several methods designed to address these problems with current TCP implementations have been proposed. SACK (Selective Acknowledgement) has been proposed as a means of addressing the loss of performance in the presence of multiple packet drops per window by selectively acknowledging packets as opposed to doing so in a cumulative manner. Sally Floyd's High-Speed TCP algorithm proposes a method of setting the TCP congestion window that is better suited to high-speed applications. This laboratory examines the efficiency and practical aspects of these two methods that address these issues.

2 Objectives

The first objective of this laboratory will be to examine the impact of end-to-end delay and data rate on the duration of the slow-start and congestion avoidance phases of TCP Reno. The second objective will be to examine the impact of end-to-end delay and data rate on the performance of SACK TCP relative to the on-SACK TCP Reno implementation. The next goal will be to examine the efficiency of high speed TCP, as recommended by Sally Floyd, in a situation where the link can be considered to be in the "high-speed" regime in order to fully take advantage of the modified algorithm. The last goal will be to examine the validity of the expression given in Equation 4-1 via Internet experiments involving downloading from various servers with different packet loss rates.

3 Preparation

The current congestion control algorithm in TCP sets the congestion window size (CongWin) such that it limits the maximum segment size (MSS). The size of the congestion window is dynamically adjusted based on perceived congestion in the network. When a TCP connection begins, the window size starts off at 1 MSS. In order to take full advantage of the link capacity, the window size increases exponentially rather than linearly. Once CongWin reaches a threshold value Thresh, it stops the exponential increase and increases linearly in size until a loss event. Once congestion is detected via a triple duplicate ACK, it halves the window size and eventually re-increments in a linear fashion. The initial exponential increase is known as **slow start**. In this portion of the lab, we are asked to determine the effect that propagation delay and data rate have on the duration of the slow start phase.

The TCP algorithm currently acknowledges proper receipt of packets in a cumulative manner: an acknowledgement for packet N implies that all packets with

sequence numbers N and below have been properly received. This method of acknowledgement can lead to severe performance degradation upon multiple packet losses, as the sender will decide to resend an entire window of packets upon a loss event. Many of the packets in this window of data may, however, have simply been delayed in the network and will thus be unnecessarily re-transmitted. A more efficient method of acknowledging packets would be to selectively do so (SACK), hence minimizing the number of unnecessary re-transmissions on the part of the sender. For this part of the lab, we are asked to determine the relative performance gain of SACK over non-SACK TCP as a function of link delay and data rate.

Another problem with the current TCP implementations is that the congestion control algorithm being used would require that an extremely low packet loss rate be achieved for a large CongWin value. This ultimately limits the maximum throughput that can be achieved by a TCP connection in a realistic environment. An alternative algorithm, proposed by Sally Floyd[1], seeks to alleviate this limitation on CongWin, and hence the maximum achievable throughput for a typical networking environment. The high-speed algorithm uses a different formula that the one typically used for computing the current CongWin value as a function of the packet loss over the link. Thus, in this part of the lab, we will assess the efficiency of the high-speed algorithm in terms of its fairness in bandwidth sharing between 2 competing high-speed connections.

4 Methodology

4.1 TCP slow-start and congestion avoidance

In order to fully qualify the effects of propagation delay and data rate on the length of the slow start phase, we used the OPNET modeler. We used the built-in TCP project. We then chose the *Congestion Size of TCP Reno* scenario. We then modified properties of the link between the client and the server in the scenario, in order to adjust the delay and data rate of the line. Finally we graphed the size of the congestion window as a function of time, over a period of 2 minutes, in order to view the effects of the controlled variables.

We varied the delay from 0.10s to 0.16s until we saw a trend forming. The reason that we chose this particular range is because we found that significantly above 0.16s, the delay became so large that we saw a saw tooth trace forming. Similarly, significantly below 0.10s, the graph appeared to remain in slow start phase and then leveled off once the congestion window size met the maximum capacity of the line.

In order to best capture the effects of data rate on congestion window size, we varied the data rate from DS1 (1.544 Mbit/s) to SONET/OC-24 (1.244 Gbit/s). We used the North American standard transmission rates. We began at a DS1 rate because it is the basic transmission rate and we decided to stop increasing at OC-24 because our results at higher speeds were displaying a trend.

Finally, in order to measure the length of the slow start phase in both tests, we exported the graph of congestion window size as a function of time into Microsoft Excel. We then used Excel to accurately read the point where the window size stops growing exponentially.

4.2 TCP SACK (Selective Acknowledgement)

4.2.1 Impact of end-to-end delay on TCP performance

The *Reno_with_one_drop* and *SACK_with_one_drop* scenarios in the TCP OPNET project were opened and run by setting the number of packets dropped to be 5 (more precisely, the first 5 packets of each frame are dropped). The following congestion window size was obtained:

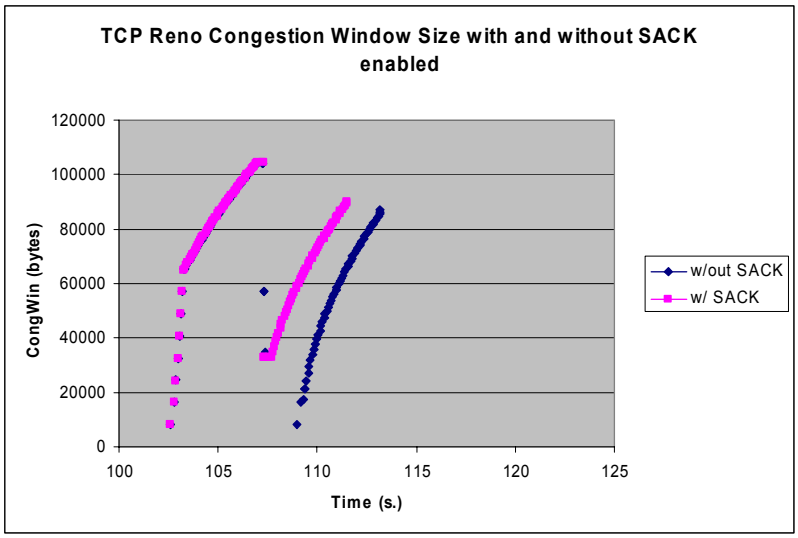


Figure 4-1 – Comparison of SACK and non-SACK TCP Reno

The same scenarios were then run with an increased end-to-end delay of 100 ms, 750 ms and 1.0 s to gauge the impact of connection delay on TCP performance. The plots of the congestion window size are shown below for all 3 cases:

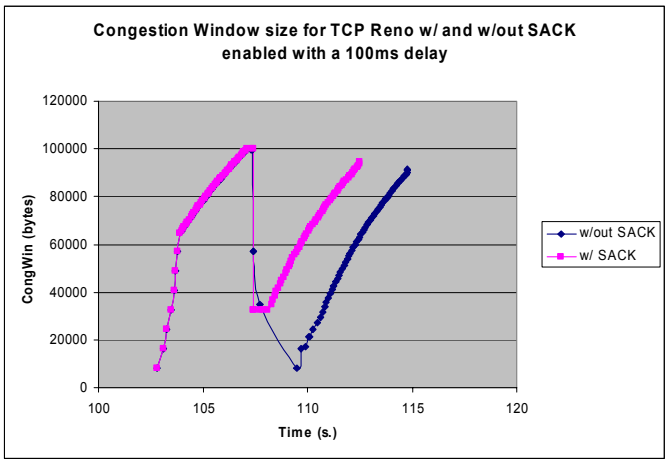


Figure 4-2 - Comparison of SACK and non-SACK TCP Reno with an end-to-end delay of 100 ms

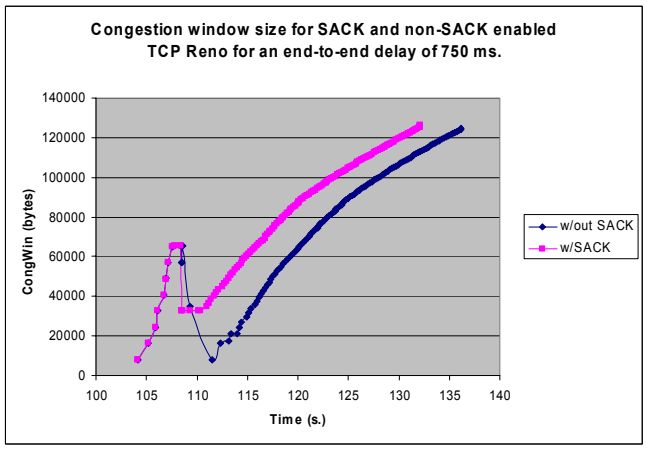


Figure 4-3 – Comparison of SACK and non-SACK enabled TCP Reno for an end-to-end delay of 750 ms.

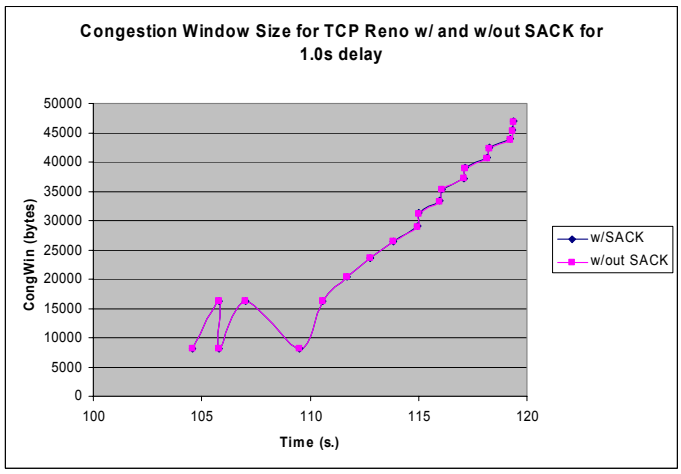


Figure 4-4 – Comparison of SACK and non-SACK enabled TCP Reno for an end-to-end delay of 1.0 s.

4.2.2 Impact of link data rate on TCP performance

The scenarios were then run with data rates of DS0 and T1 to gauge the impact of data rate on TCP performance. The plots of the congestion window size are shown below:

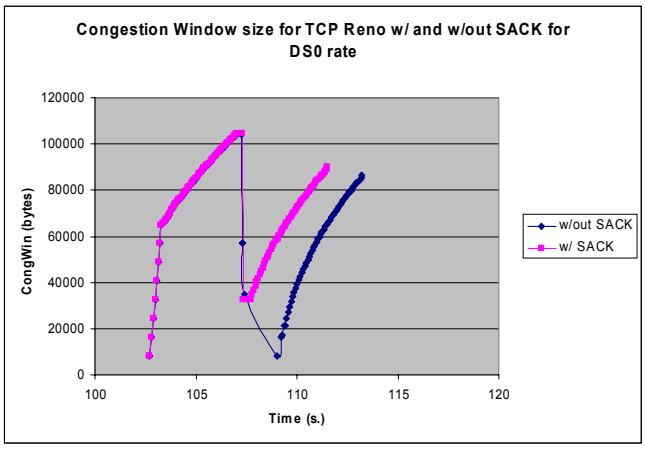


Figure 4-5 - Comparison of SACK and non-SACK TCP Reno for a DS0 data rate

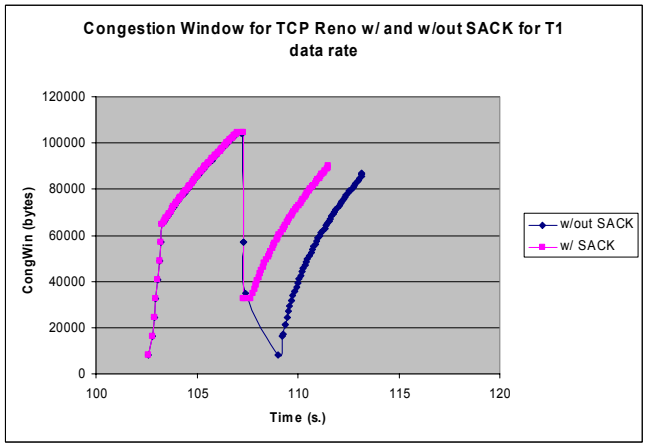


Figure 4-6 - Comparison of SACK and non-SACK TCP Reno for a T1 data rate

4.3 High Speed TCP

Using the OPNET TCP project modified to take advantage of the high-speed TCP (HSTCP) algorithm (as recommended by Floyd[1]) via the adapted process models *tcp_conn_v4* and *tcp_manager_v4*, two competing and concurrent connections were set up. The TCP Reno algorithm was used, along with a default DS0 data rate and distance-dependent end-to-end link delay. The 2nd one was set to start 0.001 s. after the 1st had started. In order to gauge the real performance of HSTCP, the link usage was set to a high load. This entails a more sustained link usage and is representative of the operating conditions that HSTCP is designed to meet. Plotted below are the bandwidth distributions as well as congestion window sizes for the 2 connections.

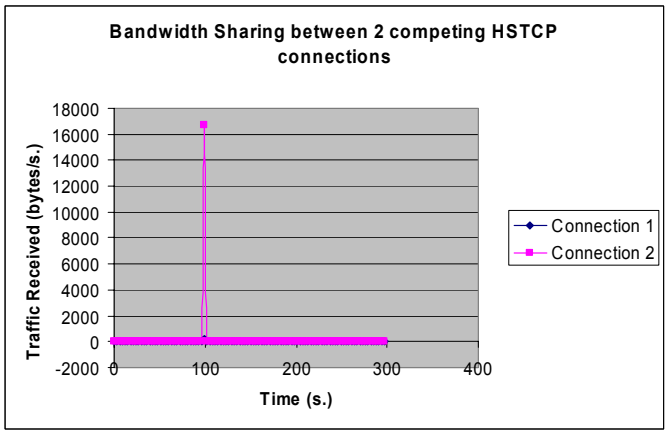


Figure 4-7 - Bandwidth Sharing between 2 competing HSTCP connections

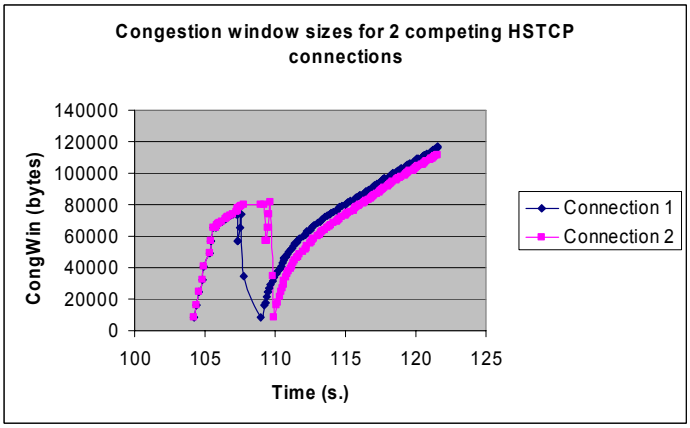


Figure 4-8 - Congestion window sizes for 2 competing HSTCP connections

4.4 TCP connections

This part of the lab was to be performed using TCPDump. The objective was to engage in a few "long" TCP connections in order to verify the validity of the *Steady-State TCP throughput* equation (Equation 4-1).

In order to engage in long TCP connections, we downloaded the same 46.8MB file from the Tucows website from 2 different locations and a 18MB file from another location. The motivation behind using this website is that we can choose the mirror site from which we wish to download. This effectively allows us to download from a specific country and compare the results. We downloaded the 46.8MB file from Israel and Hong Kong, and then downloaded the smaller 18MB file from Trinidad & Tobago. The continental distribution will adequately diversify the results.

Once TCPDump had filled our file with the log of all TCP transactions, we needed a way of counting all triple duplicates and timeouts in order to count all lost packets. This was a difficult task since the log file contained over 50,000 TCP transactions! In order to automate the process, we parsed the file into Microsoft Excel. This then allowed us to use a Visual Basic script for counting purposes. We identified a triple duplicate by always comparing a particular ACK sequence number with the last three. Once we found four consecutive acknowledgements of the same sequence number, we knew that a triple duplicate had occurred and one packet had been lost. Furthermore, identifying a timeout was not as straightforward. We realized that the only time a server should be resending information is if a triple duplicate or a timeout has occurred. Hence we setup the script to certify that the packet indexes were increasing as time progressed, in the event that there was a decrease in the packet index, a packet was resent, we simply conducted a triple duplicate test. If a triple duplicate had not occurred, we knew there had been a timeout. This however became very complicated when tracking timeouts for a connection that opened two or more ports in parallel, because then all sequence numbers were specific to the port that they represented. We then had to modify the script to keep track of the last acknowledge sequence number for each port in use. The script can now track up to 5 simultaneous ports. We used this script to analyze the log files of our 4 transactions.

$$B(p) = \min\left(\frac{W_{\max}}{RTT}, \frac{1}{RTT\sqrt{\frac{2bp}{3}} + T_0 \min\left(1, 3\sqrt{\frac{3bp}{8}}\right) p\left(1+32p^2\right)}\right)$$

Equation 4-1 – Steady-State TCP throughput

Above is a script that calculates *B(p)*, the throughput, W_{max} is the maximum congestion window size (as advertised by the receiver buffer), *b* is the number of packets acknowledged by a received ACK, T_0 is the timeout period, and *p* is the probability that a packet is lost given that it is either the first packet in its round or the preceding packet in its round is not lost. We needed to satisfy the validity of this equation for all four trials. The parameters in this formula were computed as follows: *b*= 2 since for all three trials, during the majority of the time, 2 packets are acknowledged with one ACK. The error introduced by this assumption is minimal since close to 95% of the time, there are 2 packets per group. If this is not the case, there is only a maximum of 3 and a minimum of 1 per group. Hence, the assumption is sound. The W_{max} is the maximum

window size of the receive buffer in packets. This value (in bytes) can be read directly from the log file of the TCPDump log line, it is the `win` parameter as described in experiment 1. In order to transform this into packets, divide by the number of bytes per packet. This value is also indicated in the log file. After a packet is sent, the bytes sent are included in brackets. *RTT* is the average round trip time of a packet. This value is quite complicated to calculate because the TCPDump is only reading packets received by the client. So all the timing values are the times the packets were received and not the time they were sent.

There are 2 basic ways of finding the correct round-trip time. The first is from the very first line of the log, assuming logging has begun at the beginning of the connection. At this point, we know the congestion window is of size 1, as stipulated by the TCP slow start phase. Hence the round trip would be the difference between the time that the acknowledgement was sent by the client for that one packet and the time that the next one was received. This works because we know that when the congestion window size is 1, the server must wait for the acknowledgement of that 1 packet before sending out the next. This is the same reason why this method cannot be employed anywhere within the log file. We have no way of tracking the server's congestion file window size aside from physically counting all acknowledgments and timeouts/triple duplicates. At any given time, the server will most probably have a congestion window size above 1. This means that the server sends out multiple packets without waiting for an acknowledgment. The *RTT* therefore cannot be determined. The second way of finding the round trip time from the TCP log is from the occurrence of a triple duplicate. When a triple duplicate occurs the client has not received a specific packet, the sever must then send back one packet, and one packet only. We can then measure the round-trip time from this as the difference between the time that the fourth packet in the triple duplicate was sent and the time that the missing packet was received by the client. The only flaw with this technique is that the client sends out more than 4 duplicate acknowledgements. For the purpose of this lab we will assume that the server actually resends the packet when it receives the fourth – in effect, assuming none of the triple duplicate packets are lost on the way. For our purposes, we will use the second method since it is less crude and less susceptible to error.

The value of *p* is the probability that a packet is lost given that it is either the first packet in its round or the preceding packet in its round is not lost. Effectively, *p=(Total packets lost) / (Total packets ACK'ed)*. There are two ways of determining the total packets acknowledged. Once again the first method is slightly more crude, it involves dividing the total file size by the size of a packet, implying that all packets have been acknowledged. We have actually trained our script to count all acknowledged packets, as a more precise method. Finally, *T* is the average duration of a timeout. In cases, such as ours, it is possible that a timeout is not perceived. In this case, we will be using a theoretical value for this, which applies to all TCP connections and is deduced from the following formula in *Kurose & Ross*.

$$Timeout\ Interval = EstimatedRTT + 4*DevRTT$$
$$EstimatedRTT = 0.875*EstimatedRTT + 0.125*SampleRTT$$
$$DevRTT = 0.75*DevRTT + 0.25|SampleRTT - EstimatedRTT|$$

Explanations of these equations are beyond the scope of this experiment however we will mention that the *EstimatedRTT* is an exponential weighted average. The RTT calculated from the triple duplicates will be used as SampleRTT. Once we calculate all these values, we can substitute them into the formula to see if the calculated throughput is the same as that displayed by the Microsoft Windows download window.

5 Results

5.1 TCP slow-start and congestion avoidance

After simply plotting the length of the slow start phase versus the delay set on the line, we immediately noticed a trend. The length of the slow start phase is dependent on timeout. Once the first timeout is perceived, the phase ends. Furthermore, timeouts are dependent on received acknowledgements. As we increase the link delay, it takes longer for a delay to arrive back from the receiver. Therefore the connection must wait a slightly longer time for each round for the acknowledgement to arrive, thereby increasing the overall slow start duration. The data values can be viewed in the figure below. It is worthy to note that although it may seem tempting to increase the link delay in order to maximize the slow start phase's exponential bandwidth growth, it is not a good idea. The steady state effect will be that the overall process becomes significantly slower due to the longer delay.

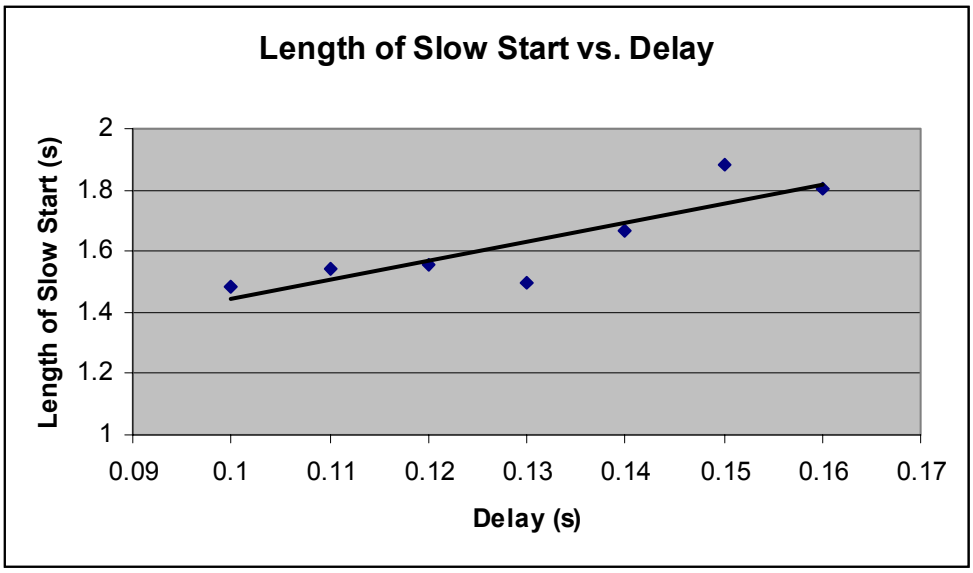


Figure 5-9 – Length of Slow Start phase vs. delay

Once all the data for the connection speeds had been collected, we plotted the graph of the length of the slow start phase vs. bandwidth. The graph can be viewed below. Clearly, the length of the slow start phase is minimally affected by the change in data rate, especially at the higher speeds. Our explanation for this effect is simply that at such high speeds, other rate controlling elements in the TCP setup, such as queues, become the bottleneck. To this effect, the link does not even end up transmitting at the specified rate.

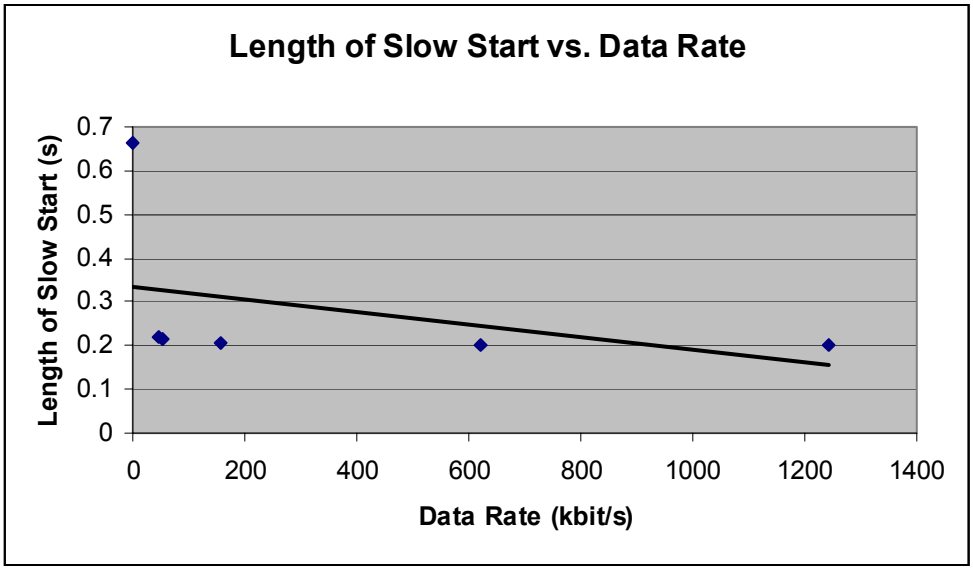


Figure 5-10 – Length of Slow Start phase vs. data rate

5.2 TCP SACK (Selective Acknowledgement)

5.2.1 Impact of end-to-end delay on TCP performance

As can be seen in Figure 4-1, though both implementations are identical in their slow-start/congestion avoidance phases, the SACK implementation of TCP Reno starts its recovery phase earlier than that of non-SACK TCP. The presence of multiple packets dropped per transmitted window of data has caused the non-SACK TCP implementation to time out, since the sender often has to wait for a retransmit timer before recovering after a loss event[1]. As a result, SACK TCP drops its congestion window size to 32 768 bytes, whereas TCP Reno resets its congestion window to 8152 bytes, or 1 MSS. Since the congestion window size for SACK TCP is larger after a loss event than the corresponding non-SACK window size, the recovery for SACK TCP is faster than that of non-SACK TCP.

By increasing the end-to-end delay, the performance of SACK TCP is expected to lessen. From Figure 4-2 , for a moderate delay of 100 ms, the performance of SACK TCP indeed degrades as the SACK implementation takes more time to recover than it did in the previous case. This can be observed from the fact that after decreasing its congestion window size, the SACK algorithm is stalling for a slightly longer time before ramping up again. As the end-to-end delay increases to higher values of 750 ms and 1.0 s., (Figure 4-3 and Figure 4-4), the performance of the SACK-enabled TCP approaches that of regular TCP. Thus, for large end-to-end delays, there is no advantage to using a SACK-enabled TCP implementation.

5.2.2 Impact of link data rate on TCP performance

As can be seen from Figure 4-5 and Figure 4-6, the data rate does not impact the performance of SACK and non-SACK TCP, as regardless of the data rate of the channel, SACK TCP recovers faster than its non-SACK counterpart. This can be explained by the fact that both implementations operate independently (say, on identical, but distinct channels) using the same

bandwidth. As a result, the relative performance variation of the SACK-enabled TCP over that of the non-SACK TCP is negligibly small.

5.3 High Speed TCP

By simulating 2 competing connections which make little sustained use of the link bandwidth, it would appear that high speed TCP is a fair algorithm as far as partitioning of bandwidth is concerned. However, in order for the algorithm to operate in the high speed regime, both connections must be made to use the link for a prolonged time. This can be accomplished by arranging a large file to be downloaded.

Thus, under heavy load conditions, the arrival of the 2nd competing connection deprives the 1st of any fairness in bandwidth sharing. As shown in Figure 4-8, once the 2nd connection begins, the 1st connection experiences a timeout event. Shortly thereafter, the 2nd connection also times out. It is evident that both connections, in running the same algorithm, seek to maximize their data rate. As a result, both connections inevitably collide and time out. Unfortunately, after recovering, it is only the 2nd connection that receives the full link bandwidth, whereas the 1st connection is left with only a fraction of the link's resources.. Thus, high speed TCP is not perfectly fair in bandwidth sharing when operating under the "real" high speed regime in which large files are transferred over the link.

5.4 TCP connections

After conducting the TCPDump's from the three locations, we needed to calculate the appropriate parameters before plugging them into the *Throughput Equation*. The following is a sample parameter calculation for the TCPDump from Hong Kong. The script calculated 10 triple duplicates and 0 timeouts in the log file. Since it outputted the lines that these events occurred, we went into the log file and manually calculated the RTT for each triple duplicate. We used Microsoft Excel to calculate the exponential weighted average of the following 10 sample RTTs and we arrived at the following values.

Calculation of Exponential
Weighted average

Sample	Estimated	DevRTT
0.290661	0.290661	0
0.290398	0.290628	5.75313E-05
0.297856	0.291532	0.001624246
0.290919	0.291455	0.001352193
0.290395	0.291323	0.001246027
0.297638	0.292112	0.002316029
0.276571	0.290169	0.005136608
0.298063	0.291156	0.005579193
0.29743	0.29194	0.005556822
0.298097	0.29271	0.005514396

Figure 5-11 – Sample Calculation of Exponential Weighted average for Hong Kong

Estimated RTT = 0.29271 sec

DevRTT = 5.514E-3

*Timeout Interval = EstimatedRTT+4*DevRTT*
Timeout Interval = .29271 + 4*5.514E-3 = 0.314767 sec
P = (# timeouts) / (# acknowledged packets)
P = 10 / (17432) = 5.74 x 10^{-4}

W = (16944 bytes) / (1412 bytes/packet) = 12 packets

b = 2 (for all cases)

B(p) = (40.99621 packets/sec) * (1412 bytes/packet) = 57.886 bytes/sec

After calculating all the equation parameters for all three locations, we proceeded to plugging them into the equation. Once again we used Microsoft Excel. The following tables show the parameters calculated for all locations and the resulting throughput calculations follow.

	W_{max} (packets)	Packets Acked	Triple Duplicate	Timeout	RTT (ms)	Timeout (ms)
Hong Kong	12	17432	10	0	292.71	314.767
Israel	12	17448	12	0	243.764	434.176
Trinidad	12	12684	70	0	161.326	257.394

Figure 5-12 – Equation parameters for all locations

	Calculated Throughput (kbytes/sec)	Measured Throughput (kbytes/sec)	% Error
Hong Kong	57.9	52.5	10.3
Israel	69.5	74	6.1
Trinidad	100.7	89.6	12.4

Figure 5-13 – Length of Slow Start phase vs. data rate

The diversity of the results is evident from the above table. Hong Kong has the highest throughput time due the greater distance traveled by packets. Second is Israel and third is Trinidad, in order of geographical distance. Another important point is that for Israel and Hong Kong, the W_{MAX} / RTT part of the *min()* operator in the *B(p)* equation dominated. Whereas for Trinidad, the second part of the *min()* function was dominant. This is due to the fact that the triple duplicate count in Trinidad is highest while the number of packets acknowledged is lowest. Therefore, the ration between the two, $p = 0.005519$ ten times larger than the ratio for the other two. The largeness of *p* makes the second part of the *min()* function larger and thus allows it to dominate in the calculation of *B(p)*.

Finally, our calculated throughputs seem to be extremely close to the measured throughput considering Prof. Coates mentioned that we should expect an error of approximately 50-80%. In our opinions, the reason that we experienced such a small error is attributed to the fact that we used an exponential weighting function to predict timeout duration. Had we actually experienced a timeout, we would have had different value for timeout duration and possibly a less precise throughput.

6 Conclusion

TCP *Renoe* slow start phase duration has certain relationship with link latency as well as throughput. A linearly increasing latency results in linearly increasing slow start phase. On the contrary, throughput does not affect the duration much, especially above OC12.

SACK TCP is supposed to have higher performance than regular TCP owing to the fact that it incurs less unnecessary re-transmissions in response to loss events. The superior performance of SACK TCP holds provided that the end-to-end delay is reasonably low: for larger delays, there is no advantage between using SACK and non-SACK, as both algorithms will retransmit packets unnecessarily. Also, for high data rates, the performance of SACK TCP is very consistent, as queuing and buffering delays will be larger than the actual link delay and hence will be the limiting factors in determining its performance.

High speed TCP was conceived to alleviate the current constraints on the congestion window size in realistic networking environments. However, it has been shown that it is not a fair algorithm in bandwidth allocation. Indeed, for 2 competing connections with a heavy traffic load, the arrival of the 2nd connection causes both connections to collide and timeout and hence reset their congestion window sizes. It is only the 2nd connection that truly recovers, as it is the connection that received the larger proportion of the link bandwidth.

The steady state TCP throughput equation was proven to be very precise when measuring large downloads from Israel, Hong Kong and Trinidad. The only disappointment was that there were no timeouts in any of our TCP transactions. If there had been, we are almost certain that the precision of the equation would have changed.

7 Bibliography

[1] *RFC 2581*, [1] Kurose, J. & Ross, K. "Computer Networking: A Top-Down Approach Featuring the Internet.", Addison-Wesley 2003

[2] Mathis, M., Mahdavi, J., Floyd, S., and Romanow, A. "TCP Selective Acknowledgement Options"**.** RFC 2018, April 1996. ftp://ds.internic.net/rfc/rfc2018.txt

[3] Floyd, S. "Issues of TCP with SACK"**.** Technical report, January 1996. ftp://ftp.ee.lbl.gov/papers/issues_sa.ps.Z

[4] Floyd, S. "HighSpeed TCP for Large Congestion Windows"**.** http://www.ietf.org/internet-drafts/draft-floyd-tcp-highspeed-02.txt

[6] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose, "Modeling TCP throughput: a simple model and its empirical validation", Proc. ACM SIGCOMM, Vancouver, Canada, pp. 303-314, 1998

[7] *RFC 2581*, http://www.ietf.org/rfc/rfc2581.txt