6-1-2001

TCP/IP Implementation within the Dartmouth Scalable Simulation Framework

Michael G. Khankin
Dartmouth College

Dartmouth College Computer Science Technical Report TR2001-407

TCP/IP implementation within the Dartmouth Scalable Simulation Framework

Honors Thesis

Michael Khankin

Advisor: David M. Nicol

June 1, 2001
Dartmouth College
Hanover, NH

ABSTRACT

This paper discusses TCP/IP networking, and in particular, the DaSSF implementation of TCP/IP. The paper reviews the protocols, outlines the implementation design, and demonstrates some tests. In addition, some performance and memory usage analysis is performed.

We find DaSSF TCP/IP to be a viable option to the existing SSF. DaSSF TCP/IP is faster and uses less memory so we can simulate larger, more complex, models.

INTRODUCTION

Dartmouth Scalable Simulation Framework (DaSSF) is a discrete event simulator designed for the simulation of very large multi-protocol networks. It is a C++ implementation of a widely used Scalable Simulation Framework written in JAVA. DaSSF goals are to provide scalability, manage-ability, and portability to complex simulation models.

While DaSSF is a useful tool on its own, its mainstream use for network simulations has been hindered by the lack of some common protocol implementations. Thus we implement a common TCP/IP protocol suite within DaSSF. With TCP/IP implemented, DaSSF network simulation capabilities will dramatically improve. In fact, the Internet uses TCP/IP to communicate, so we can learn about Internet's behavior with DaSSF TCP/IP.

Also, DaSSF TCP/IP can simulate fairly large networks using fairly modest computational resources. For example, we will show that DaSSF TCP/IP runs at twice the speed of SSF TCP/IP using less than half the memory.

In addition, DaSSF TCP/IP is easily configurable through Domain Modeling Language (DML). Thus our implementation employs DaSSFNet (a new extension to DaSSF developed by Mehmet Iyigun) that supports DML (see Appendix A for DML parameters to TCP). In that respect it is quite similar to the SSF version.

In this paper we quickly review TCP/IP, describe the implementation design for the suite, and give an overview of the implemented features. We further show some test results that demonstrate various TCP behavior in different scenarios. Finally, we

conclude with a performance and memory usage analysis of our implementation by comparing it to the SSF one.

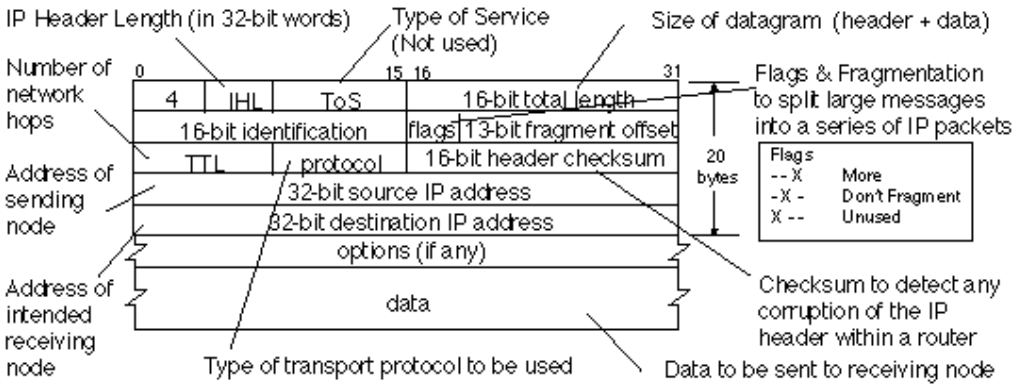
INTERNET PROTOCOL (IP)

Internet Protocol (IP) is a network layer protocol used in the Internet. IP's main features include:

- Delivery of data packets to the correct destination.
- Forwarding the data to the correct destination.
- Sending data to correct protocols on arrival.
- Prevention of packet cycling within the network forever.
- Fragmentation of the data based on physical network parameters.
- Checking data integrity

The way IP accomplishes all these features is by storing the information in the header and sending it along with the original data. A typical IP version 4 header is shown in the figure below.

Figure 1: *IP Header*



This header is typically 20 bytes long and contains the length of the packet, length of the header, 16-bit checksum, as well as the id of the protocol that used IP so that it gets delivered to the correct protocol on the other end . IP header also has various fragmentation fields specifying which part of the fragment the packet is and whether it

can be fragmented. There is also a source and destination *IP address*. IP address is a unique 32-bit number assigned to every interface in the network and is typically written as *a.b.c.d* where *a,b,c,* and *d* are the bytes in that number. When sending or forwarding a packet, IP makes a lookup in the forwarding table based on the destination IP address and gets the correct interface. The table can be set up in a number of ways. It could just contain a default interface (as is the case with a simple one interface host), or it could be set up by some routing protocol such as OSPF (Open Shortest Path First).

In case there is a cycle in the network, to prevent infinite cycling of a packet IP provides a time to live field in its header (TTL). TTL is just an integer that starts at some specified value (usually 64) and gets decremented at every hop in the network. Once TTL gets to 0 the packet is dropped.

IP IMPLEMENTATION DETAILS

DaSSF IP, similar to SSF, implements only a subset of the protocol. The implemented features include:

- Sending of packets on the right interface based on destination IP.
- Correct and efficient packet forwarding.
- Sending data to correct protocols.
- Prevention from cycling the network forever using TTL.
- Sending messages using real IP header.

Thus DaSSF IP does not support fragmentation or the checksum.

One thing to notice here, is that DaSSF IP communicates by sending real IP headers rather than just pointers, thus making it a little different from SSF. Also the byte order in the integer fields is not checked to be in any particular order and is now

architecture dependent (big vs. little endian). Thus if one were to use DaSSF IP to communicate on a real network, one would have to add an appropriate check.

Overall, DaSSF IP implementation is very simple and fast. It works well for the simulation purposes and has been extensively tested.

TRANSMISSION CONTROL PROTOCOL (TCP)

In real networks data sometimes gets lost, corrupted, or misdirected. Routers and hosts sometimes go down or crash. Occasionally power goes out. When we send data, however, we would like to know that it gets to the other end reliably and in the correct order. Thus we would like an abstraction of some sort that establishes a reliable link over an unreliable medium.

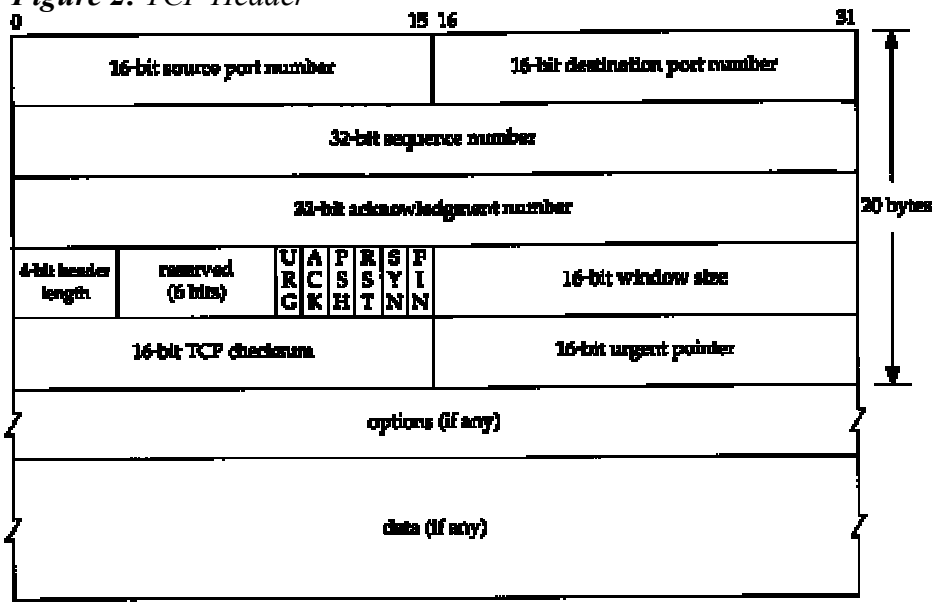
Secondly, when communicating, we would like to make different connections between any two hosts. For instance, we might want to look at a web page and at the same time establish an FTP session to the same machines. Thus we need some sort of a facility that will sort out the packets between the two sessions, thereby making sure that our FTP data gets to the ftp program while the web page will go to the browser.

TCP's function is to fulfill the needs outlined above. More specifically, TCP features include:

- Full duplex communication.
- Support for simultaneous connections and disconnections.
- Commitment to full data transmission before close.
- Reliability
- Dynamic adaptation to the changing network conditions.

To accomplish the above tasks, TCP uses its own header (Figure 2). Just like the IP header, TCP header contains various fields that allow the protocol to exchange information.

Figure 2: *TCP Header*



The 16-bit port numbers combined with the two IP addresses uniquely identify each TCP connection. Thus one cannot connect from the a given interface and port to a given remote interface and port more than once. That restriction makes sense, since otherwise TCP would have no way of knowing which sessions the packet belongs to. Also, there are some well known port numbers.

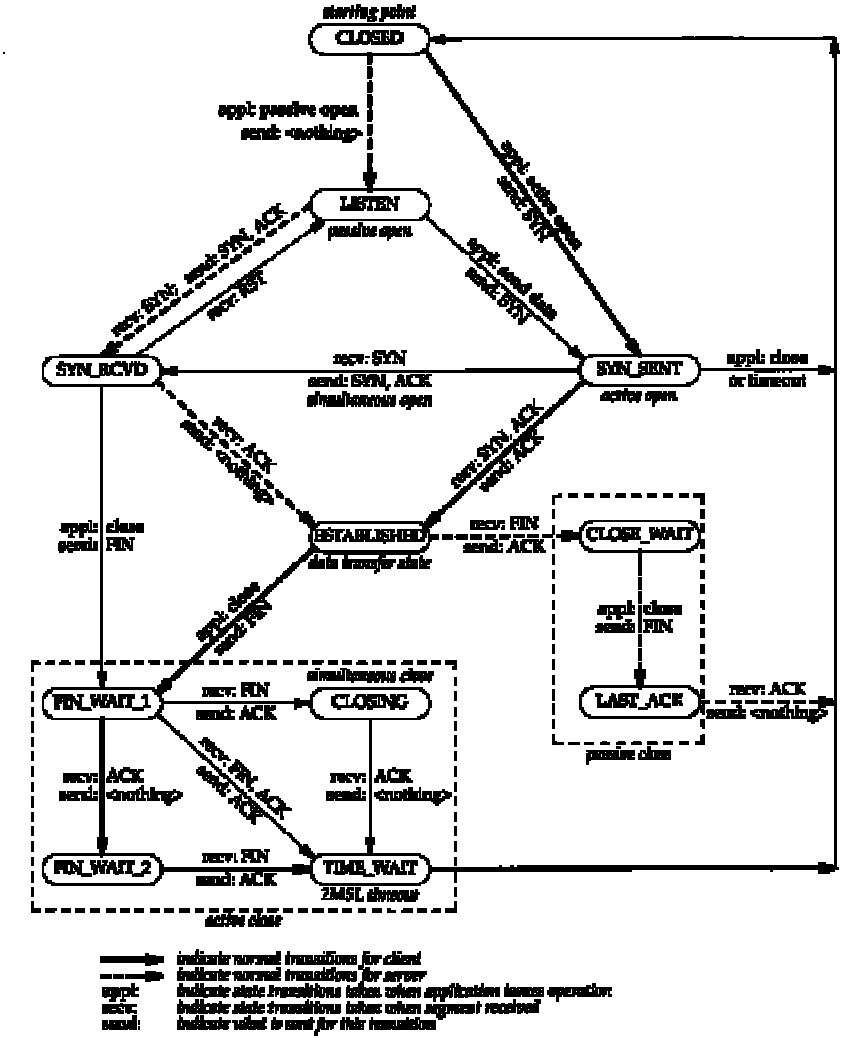
Each byte of data in a TCP connection is counted with a sequence number. For instance if we sent a packet starting with sequence number 0 of size 1024, our next packet would start at sequence number 1024. Acknowledgements work in a very similar fashion. An acknowledgement of 1024 means "I've received all the bytes up to sequence number 1024." Acknowledgement field in the header is only valid if the ACK flag is turned on. Every byte needs to be acknowledged as being received.

Other fields in the TCP header include the SYN and FIN flags, which are used to establish and break a connection. There is a RST flag which is sent when something goes wrong with the connection. Finally, there are PSH and URG flags that specify whether the data should be pushed to the user and if there is urgent data included.

Finally, another important field in the TCP header is the window size that the receiver advertises to the sender. It specifies how much data the receiver is willing to accept.

The state transition diagram for TCP is shown in Figure 3.

Figure 3. *TCP State Transition Diagram*



The connection is established via the 3-way handshake. Both sides have to send their respective connection requests, and wait for them to get acknowledged by the their peer. Disconnection is done in a similar fashion.

TCP provides reliability through its elaborate timeout mechanism. When the timeout occurs, data is resent. For every consecutive retransmission, the timeout value doubles (it is bounded at 64.0 seconds). Such a behavior is called *exponential backoff*. When a certain number of consecutive timeouts happen connection is considered dead and dropped.

TCP uses an adaptive timeout scheme. It tries to estimate the packet round trip time to its peer by measuring how long it takes for a packet to be acknowledged. When a the packet acknowledgement arrives for the packet being measured it updates it's internal variables using Jacobson's algorithm (1988):

$$Err = M - sRTT$$
$$sRTT = sRTT + gErr$$
$$DEV = DEV + h(|Err| - DEV)$$

Where M is the measurement of the round trip time, sRTT is the round trip time estimate, and DEV is the smoothed mean deviation of the round trip time. *g* and *h* are constants typically set to 1/8 and 1/4 respectively. A retransmitted packet's measurement is thrown away since we don't know if the acknowledgement came for the first or later retransmitted packets. That is called Karn's algorithm.

When first measurement arrives we have no estimates for *sRTT* and *DEV* yet so they are typically initialized as follows:

$$sRTT = M + T$$
$$DEV = sRTT / 2$$

Here T is the TCP slow timer interval (typically 0.5 seconds). TCP timeout value, *RTO*, is then typically set to *sRTT + 4DEV*.

Thus we know how TCP sends data, manages connections, and provides reliability. Another important piece of TCP is adjusting the amount of data sent according to network conditions. The protocol achieves this goal by estimating the amount of congestion in the network. TCP maintains a variable called the *congestion window* and makes sure never to send more data than the minimum of the congestion window and the remote advertised window. In addition a *threshold* variable is maintained that is used to determine how fast the congestion window grows. Initial threshold is typically 65,536 bytes.

Congestion window varies inversely with the amount of congestion in the network using the number of retransmits as a proxy for congestion. Congestion window does the following on the acknowledgement of a segment:

$$\begin{aligned} &\textit{if } CW < Threshold \\ &\qquad CW = CW + MSS \\ &\textit{otherwise} \\ &\qquad CW = CW + MSS^2 / CW \end{aligned}$$

Here *CW* is the congestion window and *MSS* is the maximum segment size that TCP can transmit. Thus when congestion window is below threshold it grows exponentially (each added segment now leads to an extra acknowledgement and segment). That stage is called *slow start*. After congestion window hits the threshold it grows roughly linearly and that stage is called *congestion avoidance*. On a retransmission threshold is set to be half the congestion window or the advertised receive window (whichever is smaller) and congestion window is reset to 1 *MSS*. Thus when retransmissions happen, TCP stops

sending a lot of data since its congestion window collapses. In fact, one can show that TCP sending rate *T* obeys the following formula (Floyd):

$$T \leq 1.5 \sqrt{2/3} * B / (R * \sqrt{p})$$

Here *B* is the packet size, *R* is round trip time and *p* is probability of packet loss. Thus we can see that *T* is inversely proportional to $\sqrt{p}$.

TCP IMPLEMENTATION DETAILS

DaSSF TCP implements the following TCP features:

- Go-back-n version of TCP
- Slow start and congestion avoidance
- Proper timeouts using "efficient" slow and fast timers
- Real and Fake data transmission/reception modes
- Persist Timer
- Fast Retransmit/Fast Recovery
- RST flag
- Proper remote receive window handling
- Delayed acknowledgments
- Simultaneous connect and disconnect

Some of these features are not implemented in SSF TCP and some of these features are implemented differently there. For instance, persist timer feature is only relevant when the receive window closes up (goes to 0) and to it makes sure that there is no deadlock by sending probes to the TCP peer to find out if the window size has changed or not. Since SSF TCP assumes that only whole packets are sent and the application receives all data immediately, there is no need for it since the window will never close up. The biggest difference between the SSF TCP and DaSSF TCP is probably the go-back-n nature of DaSSF TCP. Go-back-n means that any packet that TCP is not expecting, TCP drops instead of storing for future use. That yields different retransmission patterns between SSF TCP and DaSSF TCP. Fast Retransmit and Fast Recovery algorithms, which

essentially start retransmissions on reception of duplicate acknowledgements instead of waiting for a timeout, also need to be tested differently from SSF because of go-back-n.

Besides differing in features from SSF, we also differ in some design decisions. Our TCP is state oriented. That is, each session has a state, and that state is an object which has it's own defined behavior. That way we avoid extremely long code for some states that barely need to do any processing and thus achieve better code readability and performance. Each state can transition TCP into a different state, based on the state diagram above.

Also TCP timers are centrally implemented. That means that there is only one pair of timers per host used by all sessions. Those timers don't fire at an interval like real TCP operating system timers might do, but rather compute when the next timer would be if we had a timer of a given granularity (i.e. slow timer or fast timer). That is a big performance save.

Our TCP implementation is not without problems. Some features still need to be implemented. Some examples are:

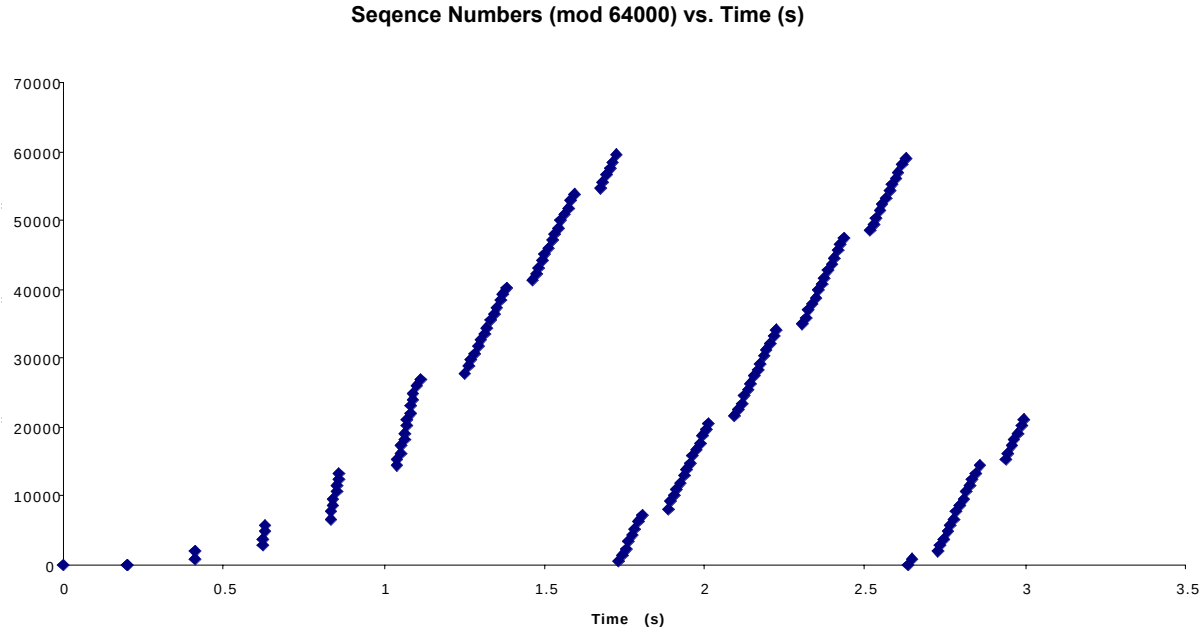
- TCP should store packets out of sequence if within window (no go-back-n)
- Silly Window Syndrome Fix
- URG, PSH flags are not implemented
- Nagle Algorithm

Silly Window Syndrome Fix and Nagle algorithm make sure that TCP does not send unnecessarily small packets. URG and PSH flags are not widely used and are thus not implemented. As we already mentioned, go-back-n receive window needs to be adjusted to be more like a selective repeat protocol that stores segments that are expected in the future if they come out of order.

TCP EXAMPLES

In this section we will demonstrate some examples of our TCP in various scenarios. Some of these are based on the TCP validation tests available from the SSF web site.

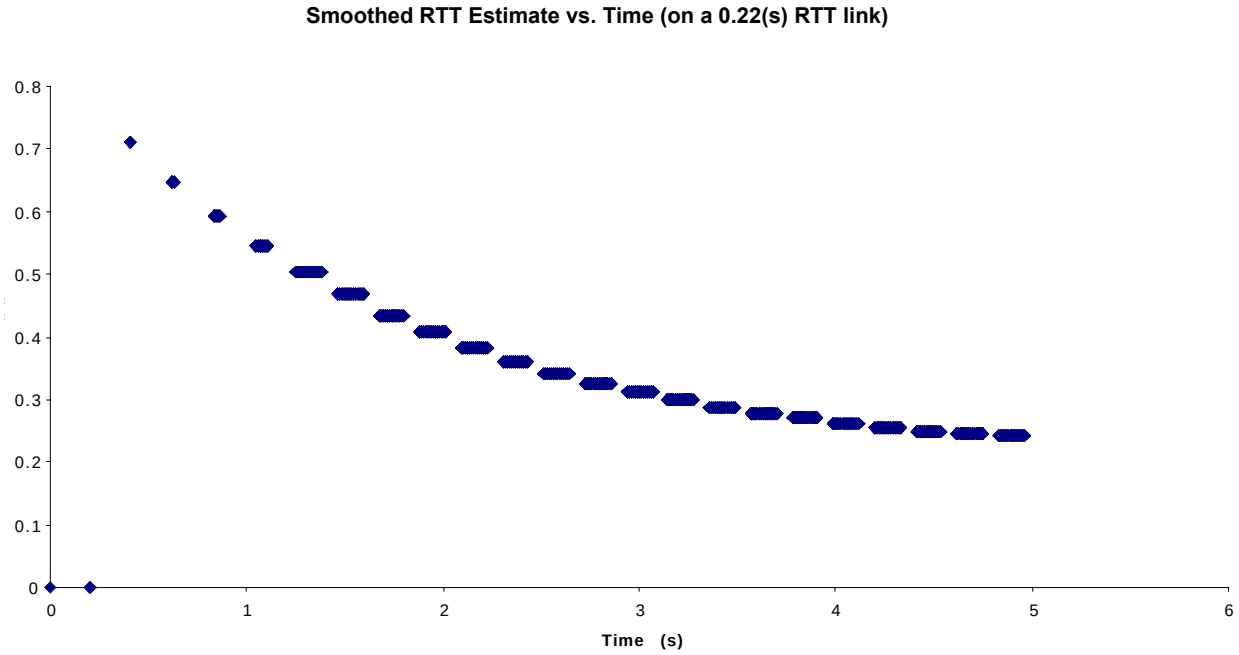
Figure 4: *Regular transmission with a send and receive window size of 13 segments*



In Figure 4 we can see the effective window growing up to 13 exponentially with congestion window. The time between burst starts is how long it takes for the first packet to get acknowledged, or the round-trip-time. Here round trip time is about 0.22 seconds. In the scenario above the application simply sends some data to the other end of the connection that keeps reading it. The first point on the graph is the SYN segment.

Figure 5 shows how *sRTT* is changing over time in the scenario of Figure 1. We can see that *sRTT* is approaching 0.22 seconds. Unlike in SSF, our measurements for packets are not done using a 500ms timer but rather getting the time when the packet sent and when the acknowledgment is received. Thus we get a much smoother shape than

Figure 5: *Round trip time estimate being dynamically determined.*



SSF TCP. Of course our timing could be changed so that calculation is done as if we were counting using a 500ms timer. We can also see initial value set to 0.22 (measured) + 0.5 (slow timer).

In Figure 6 below, we see congestion window from the previous examples. We observe slow start only since there are no retransmissions. When congestion window reaches the maximum IP datagram size (65,536) it gets bounded. We can also convince ourselves that slow start is indeed very fast – i.e. exponential. Threshold is also never changed in that example since it is initially at 65,536 bytes.

Figure 6: *Slow Start*

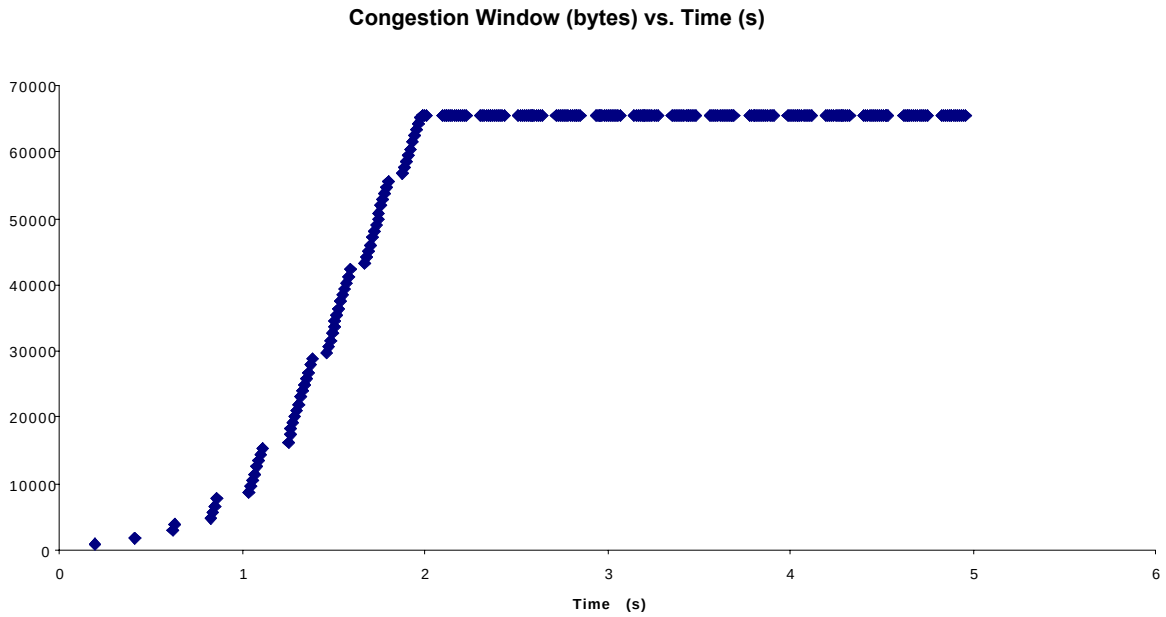


Figure 7: *Exponential Backoff*

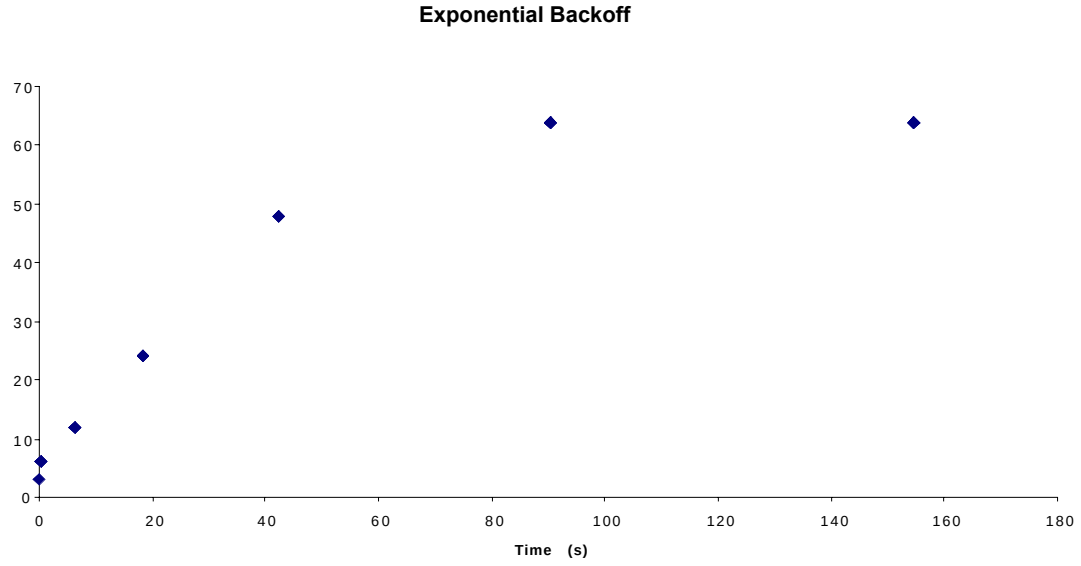
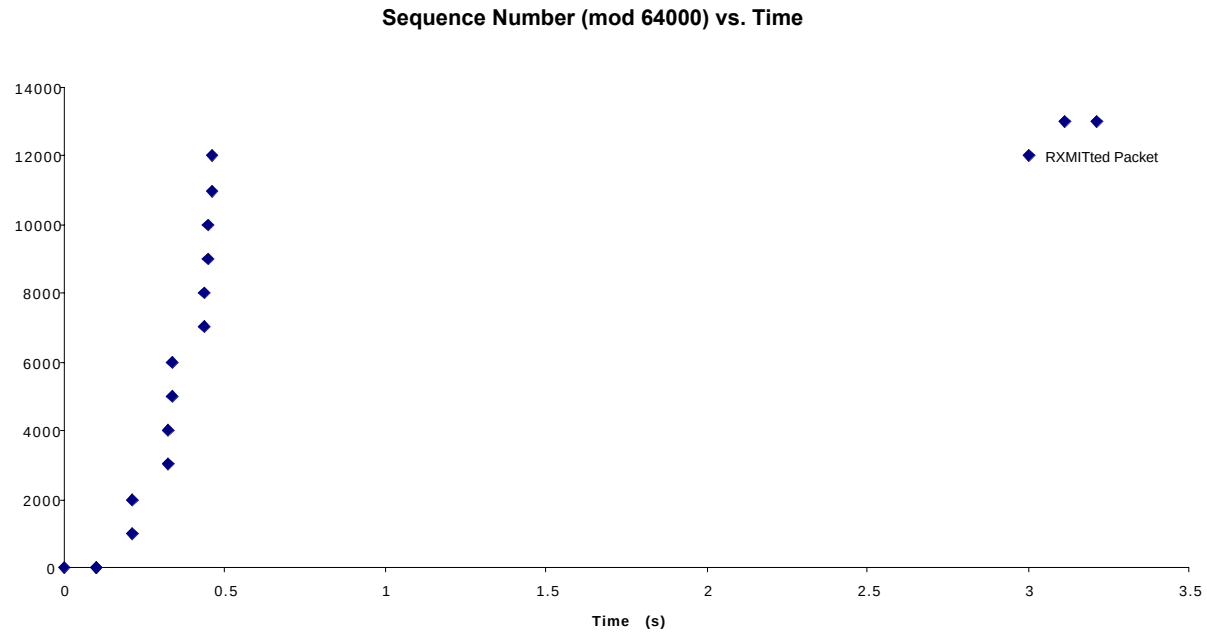


Figure 7 shows a scenario when once the connection is established, no packets can get through. Thus we see the retransmission timeout increase exponentially on every retransmit from 3 to 6, 12, 24, 48, and finally bounded at 64 seconds.

Figure 8: *Last Packet is Dropped.*



In Figure 8 we see that segment with sequence number 12001 (13th segment, last) is dropped at roughly 0.5 seconds into the transmission. We see it retransmitted 3 seconds into the simulation. The two points on the graph after it are the FIN and ACK segments closing the connection.

Figure 9 below shows a more complex example. Packet with sequence number 31,000 is dropped. There are multiple duplicate acknowledgements arriving for segments after that. Thus we see a fast retransmit happening. Finally when timeout occurs at time 3.5 (remember TCP uses 0.5 second timer) we see a go-back-n retransmission when all the segments starting at 32000 are retransmitted. (31000-31999 was already retransmitted before using fast retransmit and acknowledged). Thus we see that with go-back-n retransmissions, fast retransmit does not do much good. If we were to store all the segments after 31000, then that single retransmit would've resulted in acknowledgement of the whole window and it would no longer have to be retransmitted.

Figure 9: *Drop segment 31000, Fast Retransmit, go-back-n retransmission.*

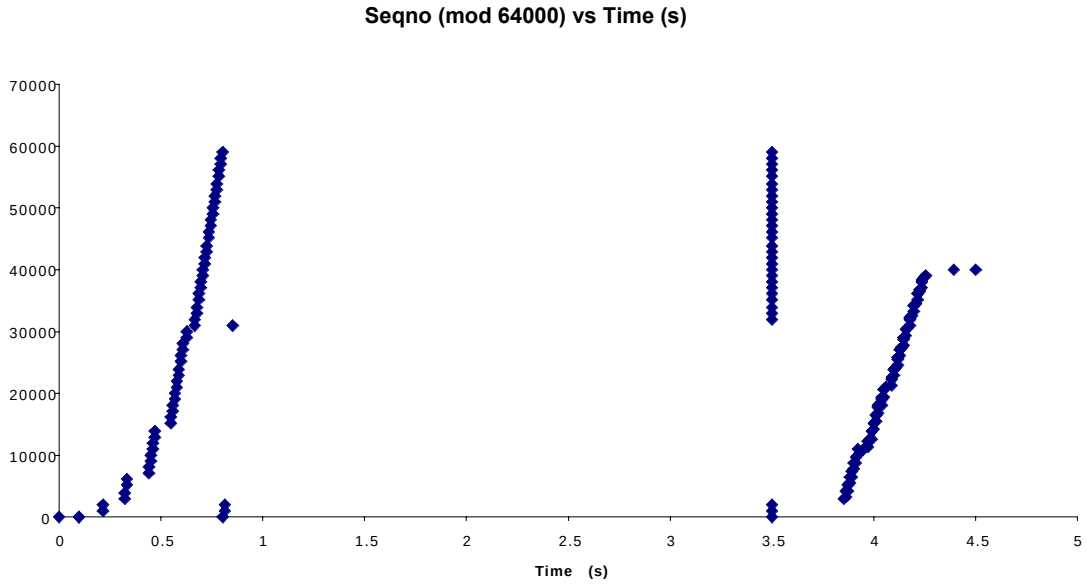
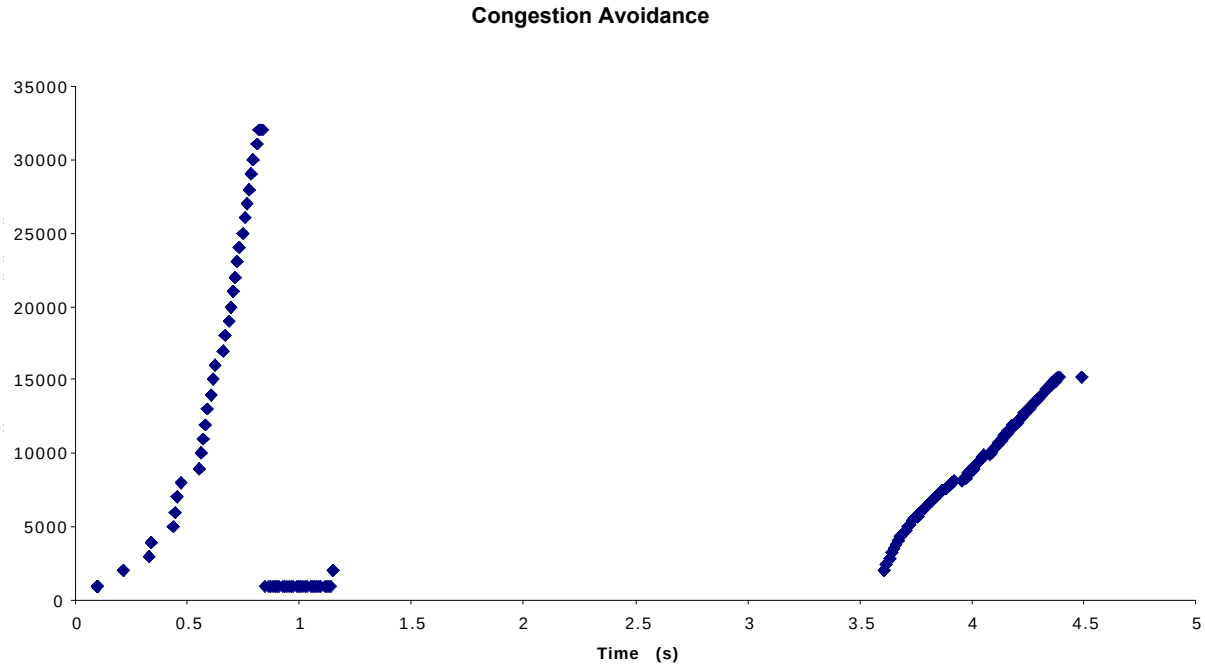


Figure 10: *Slow Start, Fast Retransmit, and Congestion Avoidance*



We see in Figure 10 what the congestion window is doing for the above example. At first there is slow start. Then, at fast retransmit time congestion window is reset to 1 MSS (threshold is set to 2MSS then). We then see the congestion increase a little bit when the segment 31000 (fast retransmitted) gets acknowledged. Then we wait until after the timeout occurs and once those retransmitted segments get acknowledged we observe the congestion window grow roughly linearly, since it is now bigger than the threshold (2MSS).

SOCKETS

User applications access TCP through what is known as the Socket Interface. Thus we have implemented the socket interface in DaSSF by making it roughly similar to real life UNIX Berkley sockets. Sockets are not unique to TCP. Other protocols use sockets as an interface (i.e. UDP). Thus when implementing sockets, we have kept in mind that other protocols might want to use them later. Therefore our sockets are designed so that one can write a protocol that uses the socket interface without ever recompiling the socket code. Therefore, sockets can be made into a library. The only restriction on that protocol would be that it support a certain interface by deriving from a certain class. The protocol should of course know how to interact with sockets by signaling (as outlined in socket documentation). The user in turn can bind to that protocol by name via sockets. DaSSF socket interface is outlined in `sock_master.h` file in DaSSF implementation.

PERFORMANCE AND MEMORY USE

Our memory and performance analysis shows that DaSSF TCP/IP works at about two times the speed of SSF and uses much less memory. DaSSF TCP/IP, unlike SSF, is also stable in the amount of memory it needs. In fact, memory optimization was a large driving force behind this project.

We have implemented SSF's tcpClient protocol in DaSSF to construct the speed and memory tests. tcpClient gets a random server, asks for a certain number of bytes, receives them, and disconnects. Then the client waits for a specified period of time and repeats the procedure.

We have tested using a 1,300 host network with 4 servers. To assure that both implementations were doing similar things we have inserted code into SSF as well as DaSSF to measure the network load (number of packets sent). We tested worst case scenario when links are perfect and network is as loaded as can be. In this case we would have no retransmissions either (since DaSSF and SSF behave differently on retransmission with DaSSF doing go-back-n).

In addition, flat (equal probability) distribution was used to ensure that things were random. While we didn't vary the seed, with increasing simulation time we expect the seed to matter less and less, thus for large times seed should have no effect if the random number generators are fairly good (which is certainly the case for both DaSSF and SSF). As already mentioned above, we measured the amount of work done by both simulators to ensure they were doing roughly the same thing.

Finally we adjusted some settings for DaSSF since it's TCP is a little different from SSF to make sure it was doing roughly the same thing as SSF. Thus all our tests

measure time taken to simulate equal sized networks for equal times, doing similar things, handling roughly the same number of events.

Thus speed and memory results are presented in Figures 11 and 12.

Figure 11: *Performance vs. Time (DaSSF vs. SSF)*

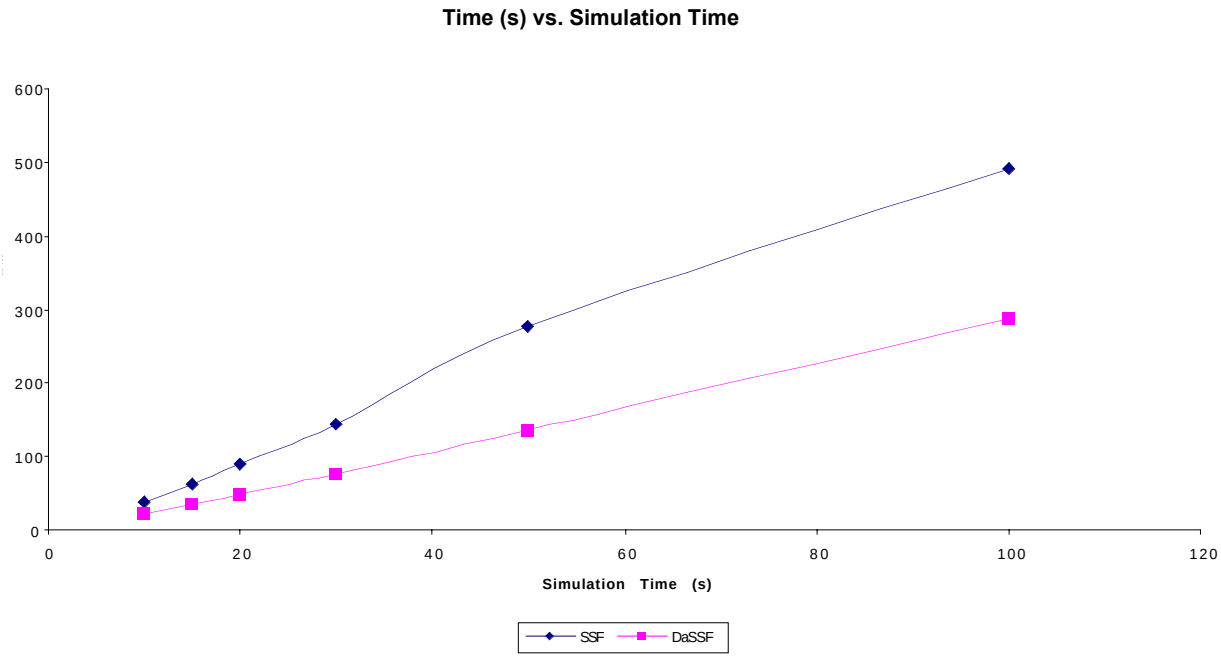
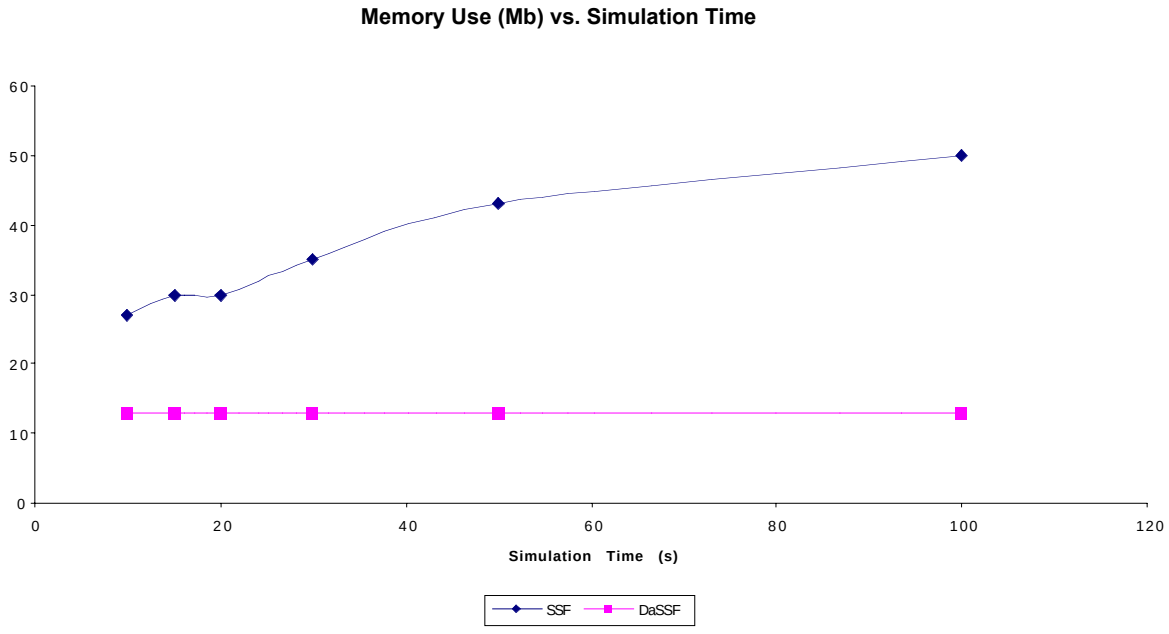


Figure 12: *Memory Usage vs. Time (DaSSF vs. SSF)*



From the graphs above we see that DaSSF TCP is faster and requires less memory than SSF. To get a sense on how much the memory requirements differ, we ran 25 connected networks such as above (i.e. 33,000 host network) on DaSSF TCP/IP and took up 340 Mb of memory on a 1GB machine. We were not able to simulate the same network using SSF since there was not enough memory for the simulation.

Thus we conclude that DaSSF is a great alternative to SSF for simulating large, complex network models.

CONCLUSION

This paper surveyed TCP/IP networking model, discussed some details of DaSSF TCP/IP implementation, reviewed some scenarios that show the behavior of DaSSF TCP/IP, and finally discussed the performance and memory usage of DaSSF TCP/IP. As a result, we saw that while DaSSF TCP/IP implementation is not yet all there, most of the work is done. A little bit of tweaking is necessary to bring the package to full functionality but the major design and implementation stages are complete. The paper also surveyed the interface to TCP and has shown that it is well designed and is ready to be used by other protocols. Finally we saw that DaSSF TCP/IP runs twice as fast as SSF and also uses a lot less memory.

Thus DaSSF TCP/IP will enable people to write protocols on top of TCP and simulate them on large complex networks without resorting to expensive computing power. In addition, it will hopefully help us learn something about the largest network of all – the Internet.

REFERENCES

Floyd, Sally and Fall, Kevin., "Promoting the Use of End-to-End Congestion Control in the Internet," IEEE/ACK Transactions on Networking, May 1999, *to appear.*

Kurose, James F. and Ross, Keith W., Computer Networking, Addison Wesley, Boston: 1999

RFC 793, J. Postel, Transmission Control Protocol, September 1981.

RFC 1122, R. Braden, Requirements for Internet Hosts -- Communication Layers, October 1989.

RFC 2581, M. Allman, V. Paxson and W. Stevens, TCP Congestion Control, April 1999.

Scalable Simulation Framework, URL: "http://www.ssfnet.org"

Stevens, Richard W., TCP/IP Illustrated Volume 1: The Protocols, Addison-Wesley, New York: 1994

Stevens, Richard W., TCP/IP Illustrated Volume 2: The Implementation, Addison-Wesley, New York: 1994

Tanenbaum, Andrew S., Computer Networking, Prentice Hall, NJ: 1996, 3rd ed.

Appendix A: DML configuration of TCP

TCP supports the following DML attributes:

ISS	# initial sequence number
MSS	# maximum segment size
RcvWndSize	# receive buffer size
SendWndSize	# maximum send window size
SendBufferSize	# send buffer size
MaxRexmitTimes	# maximum retransmission times before drop
TCP_SLOW_INTERVAL	# granularity of TCP slow timer
TCP_FAST_INTERVAL	# granularity of TCP fast(delay-ack) timer
MSL	# maximum segment lifetime
MaxIdleTime	# maximum idle time for drop a connection
delayed_ack	# delayed ack option
fast_recover	# implement fast recovery algorithm
show_report	# print a summary connection report

Time values are in seconds.
Boolean values are true/false