Microscopic Examination of TCP Flows Over Transatlantic Links

Antony Antony [a], Johan Blom [b], Cees de Laat [b], Jason Lee [b], Wim Sjouw [b]

[a] *NIKHEF, 409 Kruislaan, 1098 SJ Amsterdam, The Netherlands*

[b] *Universteit van Amsterdam, 403 Kruislaan , 1098 SJ Amsterdam, The Netherlands*

.

Abstract

Much of the recent research and development in the area of high speed TCP is focused on the steady state behavior of TCP flows. However, our experience with the first research only transatlantic 2.5 Gbps Lambda link clearly demonstrates the need to focus on the initial stages of TCP. The work we present here examines the behavior of TCP flows at microscopic level over high-bandwidth long delay networks. This examination has led us to study the influence of the minute properties of the underlying network on bursty protocols such as TCP at these very high speeds combined with high latency. In this paper we briefly describe the requirements for such an extreme network environment to support high speed TCP flows. We also present results collected using transatlantic links at iGrid2002 where we tuned various host parameters and used modified TCP stacks.

Key words: HSTCP, High Speed Long-latency TCP, iGrid2002, Long Fat Networks

1 Introduction

Grid applications in general can be demanding in terms of bandwidth requirements. A typical large scale scientific experiment involves at least two parites: a data producer and a data consumer. Many of the large High Energy Physics (HEP) experiments coming online in the near future such as LHC[16], DØ[12], CDF[11] and BaBar[10] are all excellent examples of this model of computing. In these large experiments the data is mostly produced at a few locations (the data producers) and then many reserarchers analyze this data at their home

Email addresses: `antony@nikhef.nl` (Antony Antony), `jblom@science.uva.nl` (Johan Blom), `delaat@science.uva.nl` (Cees de Laat), `jason@nikhef.nl` (Jason Lee), `wsjouw@science.uva.nl` (Wim Sjouw).

institutes and universities (the data consumers). Often the researchers are geographically separated by large distances. The throughput requirements for these experiments are high.

For example, the Europen Data Grid (EDG) roughly estimates the peak bandwidth of the network traffic that will flow over it in the year 2005 to be 8000 Mbits/sec from a single project (D0), and that this link will have to be shared among several different HEP projects, all wishing to dissimante their data. Not only does this data need to be collected from the experiment but a large part of it needs to be transferred to various locations over the network. Network architectures are evolving to meet this unprecedented demand. Herein we present research on the scalability of protocols to allow these kinds of applications to reach the required high-bandwidths on new network architectures. One way to provide this bandwidth is by provisioning end-to-end paths, called Lambdas [5], up to several Gbps. In the fall of 2001 SURFnet[18] provisioned a 2.5 Gbps Lambda between Amsterdam and Chicago to be used only for research. Initial tests showed that only increasing the speed of links, switches and routers in the path was not sufficient to obtain a throughput at or near the available bandwidth. We conducted extensive experiments on this transatlantic link both to understand how transport protocols behave and what additional requirements high speed flows, such as TCP, impose on optical networks. One architectural shift in our high-speed networking experiments was to minimize the number of routers (devices which process packets at layer 3 and above), and instead delegate packet forwarding to switching devices at layer 2 or below. Initial throughput measurements of a single TCP stream over such an extreme network infrastructure (i.e. the SURFnet Lambda) showed surprisingly poor results. This led us to further examination of the dynamics of TCP at the microscopic level to better understand its behavior. The primary motivation of this work is the demand from HEP community to obtain maximum throughput over long distance links using a single or atmost a few TCP streams. Currently in the HEP community there are several projects underway to try to over come these limitations. However, these projects focus primarly on increasing some of the default parameters of TCP (SSThreash [7], etc). Simply increasing network capacity does not always improve end-to-end performance. The exclusive availability of the SURFnet Lambda for research has allowed us to investigate this problem.

The performance issues of TCP/IP for large data transfers over high-bandwidth long-latency path is a well known problem [6]. The problem is to discover the bottleneck of a TCP flow (the slowest link in a chain of networks) between two PCs connected using a long-latency high-bandwidth path. There are several issues related to this problem: network characteristics (router, switches, slow links), the implementation of the TCP stack and specific parameters passed to the TCP algorithm by the hosts. In section 2 we examine the characteristics of the equipment and how this influences TCP and what requirements TCP imposes on the network. Section 3 briefly discusses some of the problems with the TCP algorithm on high-bandwidth long delay paths. In section 4 we discuss the effect of host and operating system parameter tuning on performance. Section 5 shows test results from different modifications to the TCP/IP algorithms and particular those of HSTCP [4] implemented by the Net100[17] project.

In the following sections we broadly classify the various stages of a TCP session into: bandwidth discovery phase (aka slow start), steady state [4], and congestion avoidance. In this work we focus mostly on the initial phase of a TCP flow, the bandwidth discovery, as we believe that this phase most influences the bandwidth obtained using TCP.

2 Properties of underlying network infrastructure

The initial configuration used for the SURFnet Lambda (2.5 Gbps) is shown in Figure 1. Two high-end Personal Computers (PCs) were connected using Gigabit Ethernet via two Time Division Multiplexer (TDM) switches and a router. The TDM switches are capable of encapsulating Ethernet packets in SONET frames up to the rate of the specific SONET channel. The hosts were connected at 1 Gbps to a first version of the TDM switch. The linecard

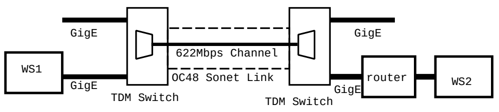

Fig. 1. Initial network setup. Two hosts connected back via two TDM switches interconnected at OC48 Link (96 msec RTT), sub channeled into an OC12.

to backplane interface posed a 622 Mbps limitation on the datapath. In a subsequent version this bottleneck was alleviated. The Round Trip Time (RTT) of the network was about 100 ms and thus a very high-bandwidth delay. Initial TCP tests conducted over this link between Amsterdam and Chicago with the first version of the TDM switch showed rather poor results. Throughput obtained using a single stream TCP session was an order of magnitude less(about 80 Mbps) than the bottleneck capacity. Tuning the TCP stack showed only marginal improvements(110 Mbps). On the other hand, a multi-stream TCP session between the same two hosts achieved a throughput of about 520 Mbps. Also a UDP stream using (`iperf`) obtained a throughput a little higher than that of a multi-stream TCP session. Note that this path was exclusively used for research so there was no possibility of background traffic to influence the results. Before shipping one of the TDM switches to Chicago we had tested a setup locally (back to back) with a negligible round trip time and the throughputs were also close to linespeed (622 Mbps). This led us to the conclusion that the problem lay in the large RTT.

The approach we took to understand the performance problem was to examine TCP at a microscopic level. A quick look at the traces showed that

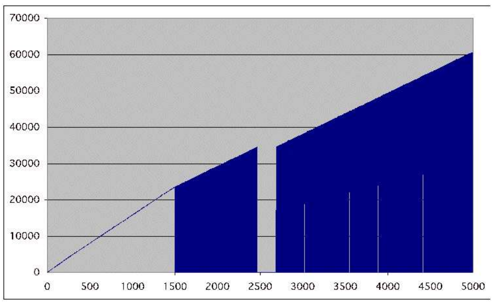

Fig. 2. relative arrival time (in μsec's) at the receiver vs packet number, for 5000 UDP packets sent using UDPMon, from Amsterdam to Chicago

TCP is very bursty in nature during the initial phase. We belive that the TDM switch could not cope with the large bursts and therefore was dropping packets. Since there is an intrinsic bottleneck of 622 Mbps for the TDM switch and the PCs are connected at a higher speed than the bottleneck, it is evident that the host will be able to overflow the switch's memory on the input linecard. In the tested configurations (hardware) flow control was not operational. This is discussed in great detail in section 3. For the rest of this section we used streams of UDP packets to simulate the behavior of a TCP burst during the initial phase.

In order to estimate the maximum possible burstsize which does not cause packetloss in the switch, we used a tunable UDP stream. The assumption here is that the burst is similar to what occurs in a TCP flow during its initial phase.

Our setup is shown in Figure 1. Two personal computers one configured as the sender (Amsterdam) and the other configured as the receiver (Chicago), were connected to the TDM switches using a Gigabit Ethernet link. The switches were then interconnected over a high-bandwidth delay product link.The RTT of the link was about 100 msec and the provisioned capacity of the link was 622 Mbps (STS12). If the sender sends a continuous stream of packets as fast as it can (about 900 Mbps, limited by PC) eventually a fraction of the packets will be dropped at the 622 Mbps bottleneck in the TDM switches.

Using `UPDMon`[20], we sent 5000 numbered UDP packets, each with a length of a 1000 bytes, as fast as possible from the sender to the reciever. Figure 2 shows the result. The horizontal axis shows packet numbers, and the vertical marks arrival times. Dropped packets get an arrival time of zero. Therefore, the shaded area under the curve indicates lost packets. The first loss occurs after 1500 packets. From then on approximately one out of every three packets is dropped. This points at the bottleneck mentioned earlier since the ratio of dropped packets agrees with the bandwidth ratio. The curve also shows that a continous block of about 150 packets is lost after this point. We assumed that these packets are dropped by the receiver. Studying the packet counters of the switches in the path supported this assumption. We believe that this is due the limitations of the receiver. The receiving PC is overwhelmed by the rate and it drops a series of packets from its input buffer. We assume the packets are being dropped while they are being copied from the memeory of the Network Interface Card (NIC) to the the memory of the reciever process. A similar kind of receiver limitation is also discussed in section 3.2. In the rest of this section we focus on an intrinsic bottleneck, namely the sender side TDM switch.

The number of packets dropped by the switch, N_d, during a burst is related to the number of packets in the burst, N_b. The speed of the incoming interface(fast), f, the speed of the outgoing interface(slow), s, and the buffer memory available at the output port of the bottleneck link, M. We assume for simplicity an average packet length of size, l. The loss can then be expressed as:

$$N_d = N_b \frac{(f-s)}{f} - \frac{M}{l} \tag{1}$$

Using equation 1, we set $N_d = 0$ and $N_b = 1500$ as that is the maximum burst which got through, and computed the available memory on the TDM switch to be approximately 0.5 MBytes.

Once we know the memory of the bottleneck and the size of burst of packets that can pass through the TDM switch we can then calculate the possible bandwidth a TCP flow can achieve during the initial phase without packet loss. We assume there are no other bottlenecks in the end-to-end path and the TCP has not encountered a congestion event.

To first order, the throughput TCP can obtain is approximated by:

$$B = \frac{f}{(f-s)} \frac{M}{R} \tag{2}$$

where B is the throughput that a TCP flow can achieve and R is the round trip time.

If we assume that TCP will try to reach a stable state where the throughput will be equal to the speed of the slowest interface, we can then substitute B=s into equation 2. This leads to a memory requirement to support a high-bandwidth delay product TCP flow as:

$$M = \frac{(f-s)}{f} sR \tag{3}$$

For the network shown in Figure 1 to support a 622 Mbps end-to-end TCP flow the minimum memory required is 3.1 MBytes ($f = 1Gbps, s = 622, R = 100msec$). From our understanding of the currently available Ethernet to TDM encapsulation devices, they do not have the required memory.

From preliminary discussions with a few vendors, we understand these devices are primarily designed for high-speed Local Area Networks (LAN) and Metropolitan Area Networks (MAN), where the RTT is small ($< 20ms$), thus negating the bursty nature of TCP flows during the bandwidth discovery phase. The problem arises when these LAN's and MAN's are interconnected to other high-speed long-latency networks and the traffic flows from these larger networks traverse the equipment in these smaller networks that were designed with LAN's and MAN's in mind, causing them to become bottlenecks. The buffer requirements for any device which has traffic flows over it that are high-bandwidth high-latency, and has dis-

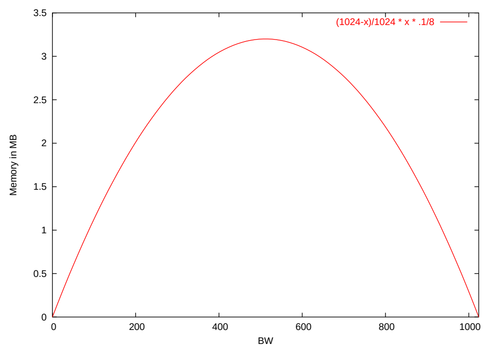

Fig. 3. Required memory at a bottleneck for an incoming speed of 1 Gbps and various output speeds for a RTT of 100 ms.

proportional interface speeds, should have enough buffer space to accommodate the difference of the input speed and the output speed of the interfaces for some large fraction of the RTT of those flows. See equation 3 for how to compute the required buffer sizes for these network devices.

Figure 3 shows the memory required in the switches to support various end-to-end speeds (for TCP) for a given RTT ($R = 100$). If we solve the quadratic equation 3 for s we can compute the TCP throughput for various values RTT for given values of M and f under the assumption there are no other bottlenecks present and the TCP flow do not encounter any othe congestion event during the bandwidth discovery phase.

$$s = \frac{f}{2}\left(1 + \sqrt{\frac{1-4M}{fR}}\right) \tag{4}$$

The result is plotted in Figure 3.

For example, using our TDM switch and a 150 ms RTT (Amsterdam to California) link, the end-to-end throughput will be about 45 Mbps. The area inside the curve is a *forbidden* area for TCP flows as packet loss may occur. The values we get using this formula also matched with the throughput obtained in TCP tests using iperf.

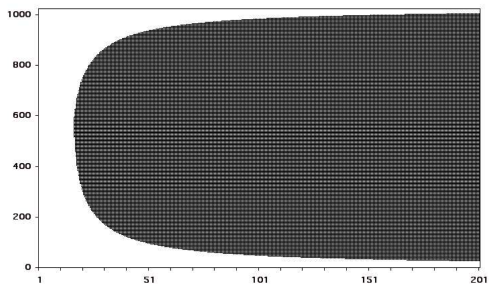

Fig. 4. Forbidden (shaded) area shows where packet loss may occur in single stream TCP flows for a given memory size of 0.5 MByte and an incoming speed of 1 Gbps. Horizontally is the round trip time of the desired destination and vertically the provisioned ”slow” speed at a TDM switch.

3 TCP

TCP is a sender-controlled sliding *window* protocol [2]. New data up to “window size” is sent when old data has been acknowledged by the receiver. The window size is limited by the host and application parameters such as socket buffer size and *Cwnd* [1]. TCP adjusts the *Cwnd* dynamically using different algorithms depending on which phase the flow is currently in. We will focus here on understanding the slow start phase. We have tested various modifications to the congestion avoidance algorithm and presented some of the results here.

3.1 Bandwidth discovery phase (slow start)

This is the initial phase of a TCP flow. After the protocol handshake [8] the sender will try to discover what the available bandwidth is so that it can compute the correct value for *Cwnd*. This discovery is done by injecting data into the network until a congestion event occurs. Fast convergence and accuracy of bandwidth discovery has a large influence on all three phases of TCP. The *Cwnd* size determines how fast a flow can reach a steady state and the stability of the flow once it has reached steady state.

If during this initial phase no congestion events are generated, the *Cwnd* effectively doubles every RTT. Thus a flow should only be limited by an intrinsic bottleneck (i.e. packet loss). If the limiting bandwidth between two hosts is the speed of the sending host (i.e. slow NIC, slow CPU) then the bandwidth discovery phase will always work correctly. However, if the connection between the hosts is limited by some other factor (i.e. router buffer, network capacity, etc) then the bandwidth discovery phase will fail due to the fact that the doubling of the congestion window can overrun the bottleneck buffer by a large number of packets [3], thus causing large packet loss. The packet loss can be computed using equation 1 if we know the bottleneck speed and buffersize at the bottleneck. Using limited slow start is a good solution to avoid buffer over run problems during bandwidth discovery. Limited slow start works by stopping the doubling of the congestion window after the window reaches a predefined threshold. After the congestion window reaches this threshold, it continues to open up, but at slower rate that is based on the size of the congestion window. This stops TCP from overshooting the bottleneck buffer by a large margin, and reduces packet loss that occurs when it does overflow the bottleneck buffer. Unfortunately this requires some sort of a priori knowledge of the bottleneck speed.

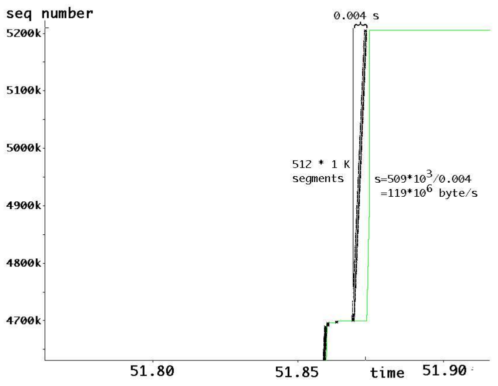

Fig. 5. Time Sequence graph showing instantaneous speed of flow

In Figure 9 we show our observation of large packet loss caused by a bottleneck. Note that after 9 RTT a burst of 512 packets leave the PC; at 10 RTT this is doubled to 1024 and this amount overruns the buffer, causing large packet loss.

The HSTCP extensions may be an excellent alternative. We believe that proposed algorithm [4] is a good starting point. In an over-provisioned network one could use a faster algorithm to increase *Cwnd*. The requirement of such an algorithm is to do bandwidth discovery as fast as possible with minimum or no packet loss.

3.2 Receiver Limitations

Typically most PC hardware exhibits the property that the receiver capacity is less than the sender for identically configured machines. This is due to the difference in overhead of sending a packet versus receiving a packet. Take, for example, two identical PCs connected back-to-back, one configured as the sender and other as the receiver. If the sender sends data as fast as possible, the receiver may not be able to keep up. When receiver is overloaded in this manner, it will start to drop packets, which in turn cause a TCP congestion event.

Figure 5 shows the instantaneous speed of a flow during slow start using a Time Sequence Graph (TSG). Notice that after 8 RTTs 512 packets leave the host. The sending host sends this data as IP packets as fast as it can. In this case the 512 packets are sent in about 4 ms, yielding an instantaneous speed close to 1 Gbps, which is line-speed. This overruns the receiver buffer and causes the flow to fall out of the bandwidth discovery phase into congestion avoidance phase. Therefore, this case is similar to that of a buffer overflow at the TDM switch as discussed in the previous subsection. A solution would be to pace out the packets in such manner that the average speed approximately equals that of the bottleneck in the path.

Figure 6 shows the combined time sequence graph of packets leaving the sender and the receiver. It clearly shows that the inter-packet delay is

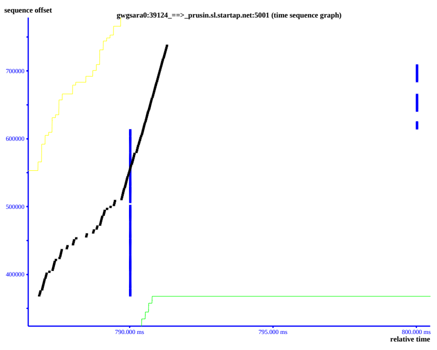

Fig. 6. combined traces from sender and receiver

very small at the receiving process. This may be due to the effects of interrupt coalescing on the NIC.

4 Host Parameters

Implementations of the TCP algorithms vary between operating systems. The behavior of TCP depends on the particular implementation and architecture of the PC, such as host bus speed, devices sharing the bus, Network Interface Card (NIC), interrupt coalescing, inter-packet delay, [20] etc. Thus using the same values as described in [19] on two different configurations can still produce varying results, especially during the bandwidth discovery phase. These differences may become less noticeable if we average these values over long periods of time.

We refer to values specific to a configuration of a PC as *the host parameters*. This also includes the TCP implementation. From our experience it has been observed that some seemingly slower hosts, in terms of CPU and bus speed, are not necessarily the slowest for TCP transfers. We assume this is due the fact that the slower hosts pace out the packets better than a faster PC, hence there is less chance for overflowing bottleneck queues in the path. The TSG in Figure 10 shows a comparison between Mac OS X and Linux 2.4.19 as sender. The data was captured at the receiver side, Linux, using `tcpdump`. It clearly shows the Mac sends packets better paced than Linux. Section 5.1 discuss the advantages of pacing the packets.

4.1 Results from tuning TX queue length

Tuning the length of the Transmit Queue (TXQ) of the sending device had a noticeable effect on the high-bandwidth high delay path. This parameter can be adjusted using the Linux command `ifconfig` with the option `txqueuelen <length>`, though one should keep in mind that the device is limited by the amount of available memory. We found that even though tuning of this variable can improve the performance of TCP by several factors, the results are not very predictable and there doesn't seem to be an easy way to precompute what the length should be. Figure 7 shows the results of testing throughput over a high-bandwidth high-delay network with several hundred different queue lengths. The default queue length is around 100 packets, while, as shown in the graph, we continued to get increased performance until about 1500 packets. Having a large transmit queue is very helpful during the bandwidth discovery phase in absence of a congestion event since it allows one to reach the maximum throughput very quickly. If a congestion event does occur, the flow will fall back into the congestion avoidance phase. This is the same reaction as if it was in steady state. The stream will then act as it normally would over a long-latency long-latency link and it will take many RTTs to recover.

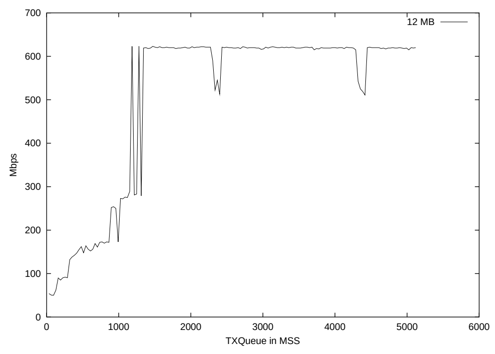

Fig. 7. TXQ in MSS vs throughput in Mbps. Steps used 32. using Net100 kernel capable of AIMD

In Figure 7 it can be seen that through adjusting the transmit queue one can clearly obtain improved throughput, but the throughput is not always very predictable. By monitoring the Web100 variable `BytesRetrans` during the tests we tried to identify packet loss to see if a failure in the bandwidth discovery phase was occurring. This was done to test if the oscillations in throughput were due to packet retransmission. However, it turned out that there were no retransmissions during the tests. We now assume that the variances may be due to the dynamics of TXQ which causes an early congestion event (i.e. premature end of the bandwidth discovery phase) and low throughput.

Net100 has coded a workaround in the Linux TCP implementation to obtain an effect similar to tuning the transmit queue. This is done by brute force, where the influence of TXQ on TCP's *Cwnd* computation is removed from the TCP stack. Results using these modifications are show in Figure 12.

To test if host behavior varies between architectures a few tests were done using an Apple laptop (used as sender) connected at 1 Gbps. Initial results look very promising. In one case we were able to get 354 Mbps between Amsterdam and Chicago. A closer look at the traces captured from the receiver is shown in Figure 10. It clearly shows that the Apple host behaves differently than the Linux host during the bandwidth discovery phase of TCP. Apparently OSX on the Apple paces the packets better than Linux, putting a larger inter-packet delay between the packets. This could be due to slightly different implementation of TCP or differences in hardware architecture such as the NIC, motherboard, CPU, etc.

5 Modifications to TCP/IP algorithm

5.1 Pacing out packets at device level

From the discussions in the preceding sections it is clear that some sort

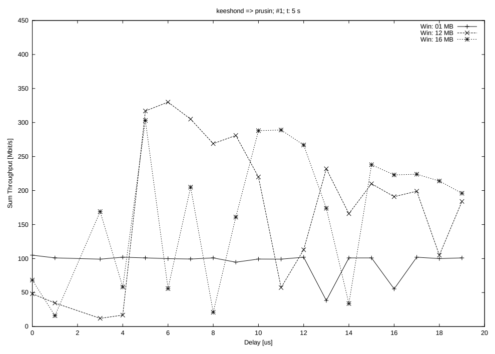

Fig. 8. Bandwidth vs. delay using iperf for a duration of 5 sec for varying socket sizes

of pacing of the packets should improve performance. We implemented a delay at the device driver level, i.e. a blocking delay of O(μ)secs. Results are shown in Figures 8 and 11. From the results we conclude that the sender should not burst packets, but try to shape the flow according to a leaky bucket algorithm. Though this may be hard to implement in the OS since it requires that the OS maintain a timer per TCP flow with μsec resolution, which could incur lots of overhead. Our initial suggestion is that future OS kernels should delegate the task of pacing the packets to NICs and allow the NIC to implement this feature at the hardware level.

5.2 HSTCP modification

The HSTCP modifications [4] discussed here are still in the development stage. Most HSTCP extensions are aimed at improving performance for steady state congestion avoidance, yet it appears that the bandwidth discovery phase may also benefit indirectly from this work. This is because if the bandwidth discovery phase ends prematurely (i.e. before

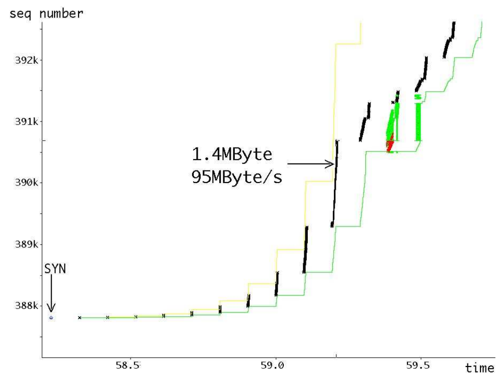

Fig. 9. Time sequence graph showing initial phase and congestion event after 10 RTTs

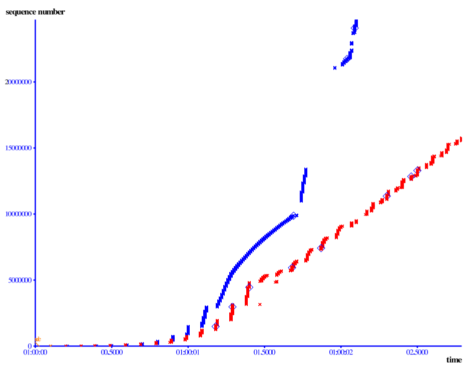

Fig. 10. TSG comparing initial phase of Linux and Mac OSX (red is Linux, and blue is Mac OS X).

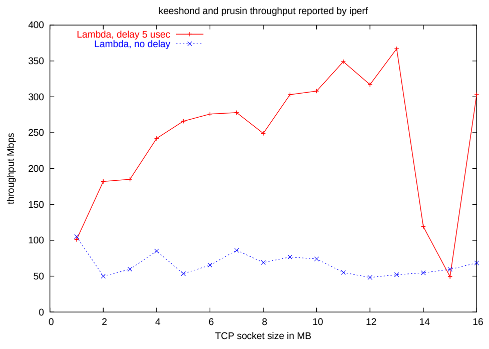

Fig. 11. Bandwidth vs socket buffer size using iperf, with delay (5 μ sec) and without delay

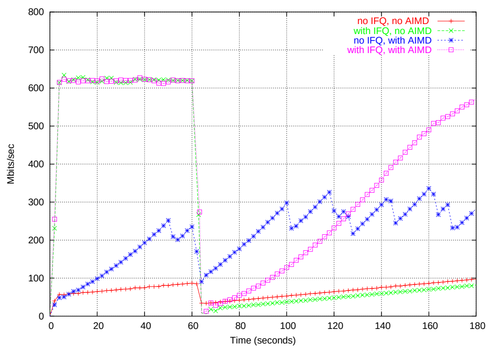

Fig. 12. bandwidth vs time using HSTCP and IFQ modifications using Net100 kernel between Amsterdam and EVL, Chicago. Congestion was introduced at 60 sec by overloading receiver

full utilization of the resources), the bottleneck utilization will be low, and then HSTCP modifications will improve utilization by ramping back up faster then traditional TCP algorithms. From the discussions above it is clear that in many cases initial stages end prematurely before completing bandwidth discovery. Using HSTCP modifications, *Cwnd* increases more slowly instead of doubling. In effect TCP flow continues to discover bandwidth more quickly, without overrunning the network buffers.

Figure 12 shows the results using HSTCP. We have run an `iperf` session for 180 seconds and created a congestion event at the receiver after 60 seconds. Congestion was created by sending approximately 800 Mbps UDP from another host to the receiver. In the first case (red line) no IFQ modifications and no AIMD [4] modifications were enabled. Due to the short transfer queue length the flow exits the bandwidth discovery phase very early, at about 80 Mbps, and then continue in steady state where it increases *Cwnd* at a rate of 1 MSS per RTT. After 60 seconds the flow encounters the induced congestion and drops to about 40 MBps and recovers from it at the rate of one MSS per RTT. In the second (green line) case Net100 was used to turn off the transmit queue congestion detection. This improves the bandwidth discovery phase, which now quickly enters high-speed steady state at about 618 Mbps. After the induced congestion event at 60 seconds it enters normal congestion avoidance phase and drops down to about 35 Mbps and recovers at the rate of 1 MSS per RTT. In the third case (pink) Net100 was again used to disable the transmit queue congestion detection of the NIC and turned on AIMD modifications in the kernel. The bandwidth discovery phase is same as in the previous test. After the induced congestion event at 60 sec the flow drops to about 40 Mbps. While recovering from this AIMD comes into effect causing *Cwnd* to increase based on the factor AI computed using values specified in [4]. Effectively the flow recovers from the congestion event much quicker. The recovery response time is dependent on the characteristics of the TCP flow, see [4] for a full description.

The fourth and last case (blue) was with the transmit queue congestion detection on the NIC enabled and AIMD turned on. Again the bandwidth discovery phase ends prematurely as in case one. Since AIMD is active and there are no more congestion events the *Cwnd* increases by the factor specified by AI. Note that there are some dips in the curve. Our understanding is that this is due to the poor default interface queue management.

HSTCP modifications and better control of IFQ clearly improves available bandwidth utilization. Figure 13 shows the bandwidth utilization (2 seconds average) of a long running running (3000 seconds) flow between Amsterdam and Chicago, USA. This test was running over a

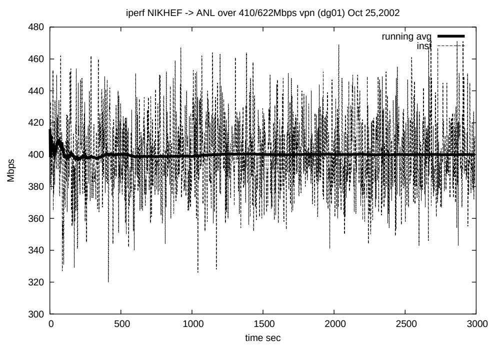

Fig. 13. bandwidth vs time for a long term iperf session between NIKHEF and ANL over 622Mbps VPN

622 Mbps transatlantic VPN with a background traffic at about 35 Mbps. The cumulative average is about 405 Mbps, while the 2 second average oscillates between 320 to 460 Mbps showing that HSTCP improves the utilization by responding to congestion events faster than traditional non-AIMD enabled TCP.

6 Conclusion

We have shown that tuning the host parameters and HSTCP are very important when trying to make best use of available bandwidth over high-bandwidth long-delay networks. The maximum throughput obtained over the Trans-Atlantic link (96 msec RTT) was 730 Mbps using a single TCP stream. Initial tests show that these modifications don't adversely affect other flows, but this still needs closer examination in an isolated environment with a large number of heterogeneous flows. Also we have shown that for the current Linux TCP implementation the specifications of the underlying network infrastructure in terms of (artificial) bottlenecks, provisioned long haul forwarding paths, queue lengths and shaping properties define the upper limit of the single stream throughput no matter how well tuned the host parameters are.

7 Acknowledgments

The transatlantic links used for conducting this research are provided to us by SURFnet, TYCO and LEVEL3. Antony Antony and Hans Blom are funded by the IST Program of the European Union (grant IST-2001-32459) via the DataTAG project. Jason Lee was supported in part by the Director, Office of Science, Office of Advanced Scientific Computing Research and Mathematical, Information and Computational Sciences Division under U.S. Department of Energy Contract No. DE-AC03-76SF00098 The authors would like to thank Richard Hughes Jones, Brian Tierney and the Net100/WEB100 collaboration for instrumenting TCP with modifications. We especially thank the iGrid2002 organizers for providing an excellent and exciting test bed.

References

[1] Mark Allman, *et al.*, “TCP Congestion Control”, RFC2581

[2] Tom Dunigan, Matt Mathis, Brian Tierney, A TCP Tuning Daemon http://www.sc2002.org/ paperpdfs/pap.pap151.pdf

[3] Sally Floyd, “Limited Slow-Start for TCP with Large Congestion Windows”, http://www.icir.org/floyd/hstcp.html

[4] Sally Floyd, S. Ratnasamy and S. Shenker, “Modifying TCP’s Congestion Control for High Speeds”, http://www.icir.org/floyd/hstcp.html 2001

[5] Cees de Laat, Erik Radius, Steven Wallace, “The Rationale of the Current Optical Networking Initiatives”, submitted for publication in FGCS special issue on iGrid2002, 2002.

[6] Jason Lee, D. Gunter, B. Tierney, W. Allock, J. Bester, J.Bresnahan, S. Tuecke, “Applied Techniques for High Bandwidth Data Transfers across Wide Area Networks”, Proceedings of Computers in High Energy Physics 2001 (CHEP 2001), Beijing China, LBNL-46269.

[7] J.P. Martin-Flatin and S. Ravot. TCP Congestion Control in Fast Long-Distance Networks. Technical Report CALT-68-2398, California Institute of Technology, July 2002.

[8] RFC 793, “Transmission Control Protocol”, Editor Jon Postel.

[9] 10Gbps, (OC192) link to iGrid2002 http://www.startap.net/starlight/ PUBLICATIONS/news-level3support.html

[10] BaBar http://www-public.slac.stanford.edu/babar/

[11] CDF http://www-cdf.fnal.gov/

[12] D∅ http://www-d0.fnal.gov/

[13] EU DataGrid, http://www.eu-datagrid.org

[14] EU DataTag,
http://www.datatag.org

[15] iGrid2002,
http://www.igrid2002.org/

[16] LHC, http://lhc-new-
homepage.web.cern.ch/

[17] Net100, http://www.net100.org

[18] SURFnet 2.5 Gbps Lambda to
Chicago, Press release
http://www.surfnet.nl

[19] TCP Tuning Guide, http://www-
didc.lbl.gov/TCP-tuning/

[20] UDPMon,
http://www.hep.man.ac.uk/ rich/net/

[21] WEB100, http://www.web100.org