FAST TCP in High-Speed Networks: An Experimental Study

Sanjay Hegde, David Lapsley, Bartek Wydrowski, Jan Lindheim,
David Wei, Cheng Jin, Steven Low, and Harvey Newman

***Abstract*— We conducted various experiments using FAST TCP on two high-speed networks and uncovered some of the hidden problems affecting high-speed TCP implementations in practice. While FAST TCP achieved good performance in many scenarios, there were also a number of cases, where adverse conditions, both host related and network related, prevented it from obtaining the expected throughput. Furthermore, tests with other protocols confirmed the existence of performance bottlenecks, some of which are still unknown to us. While various high-speed protocols such as FAST TCP will continue to improve with time, it is clear that in order to realize the vision of high-speed remote data access in Grid computing, the protocol researchers, the OS implementers, the network providers, and equipment manufactures need to understand and eliminate as many performance bottlenecks as possible. To become immediately deployable today, a high-speed TCP protocol not only needs to have robust performance under all scenarios, but also has to have the ability to provide as much information as possible to pin-point performance bottlenecks.**

***Index Terms*— Computer network performance, flow control, networks, transport protocols**

I. Introduction

The wide deployment of high-speed data networks has created ample opportunities for research collaboration across continents. Researchers frequently need to access remote resources such as computing facility and storage space across a wide area network. To meet this increasing demand, Grid computing [1], [2] has been proposed and is quickly becoming the de facto paradigm in bringing distributed resources and services together into an integrated virtual environment, where the heterogeneity of resources is masked using standardized access protocols. A critical component in the overall strategy of Grid computing is the network transport. Specifically, in order to realize the vision of distributed collaboration that involves large amount of data, there must be a high-speed network infrastructure in place. In addition, it is increasingly evident that the right transport protocol is needed in order to realize the capacity provided by such high-speed network infrastructures.

Much of the current work in Grid computing has focused on technologies and standards above the TCP layer in Open Grid Services Architecture (OSGA) [3]. There has been very rapid development in the area of Grid application, but not a lot of efforts have gone into ensuring the massive amount of data generated by these Grid applications can be shared efficiently over wide area networks. It has been shown by many that the current TCP Reno implementation often has performance problems transferring large data files over shared wide-area networks. In many high-speed networks, TCP Reno cannot achieve high throughput, even on relatively under-utilized paths.

As we have already seen in the past few years, application data has grown at a tremendous and accelerated rate, thus requiring ever increasing storage space. As technologies evolve, Grid computing will undoubtedly require and incorporate the latest state of the art network hardware such as 10 gigabit Ethernet to meet the demand of data-intensive applications. It would be imperative to address the deficiencies in the TCP protocol when doing large data transfers. There have been a number of proposals [4], [5], [6], [7], [8] that attempt to address this problem at high speed. However, there has not been a consensus on the right replacement to TCP Reno.

The key to the successful integration of a new or enhanced transport protocol is to use standard based and well structured software components that have predictable behaviors under a large number of scenarios. To this end, we believe that a transport protocol suitable for large data transfers over high-speed networks such as in Grid computing should satisfy the following requirements:

- A well structured design that can easily incorporate features that may become necessary in the future and facilitate independent upgrade of various components.
- An implementation that has both high performance and robustness under all scenarios, i.e. for both local transfer of small files as well as long distant transfer of extremely large files.

The first requirement is based on the recommendation in [9] for the need to upgrade the transport protocol component when necessary. Because of the evolution of technology, we may see unexpected combination of disparate technologies. In high-performance computing, one of the often cited performance problems is disk-to-disk transfer of large files across wide-area networks. Already, many vendors have released network interface cards with combined network protocol stack and storage controller, such as the iSCSI [10] host adapter. Thus, it's necessary and beneficial to have protocol designs that are well structured for upgrade and expansion. The second requirement is necessary in order for TCP to be a practical transport protocol in Grid computing since real networks may introduce many factors that adversely affect transfer speed. The protocol must be robust against these adverse network conditions. Furthermore, the same protocol is likely to be used to transfer small files across a LAN, or even a wireless link so the protocol must perform reasonably well under these settings.

In addition, we believe it is a good practice to have a TCP protocol that has both predictable performance and nice social behaviors under diverse scenarios. The TCP protocol should be capable of high throughput when available bandwidth permits, and it should have good network behaviors such as maintaining stable and small queueing delay and not forcing heavy packet losses.

II. FAST TCP for Predictable Performance

FAST [6] is a delay-based congestion control algorithm that aims to improve the performance of TCP Reno in the regime of high bandwidths and large delays. It departs from the loss-based congestion control schemes [11], [12], [5], [4], [8], [7]

and reacts primarily to queueing delay. In this section, we will give a brief overview of FAST and delay-based congestion control in general, and highlight some of the advantages they offer in the intended regime of operation.

A wealth of literatures (see [13], [14] for a survey) have shown that network congestion control may be interpreted as a distributed optimization problem, where sources (TCP senders), and links (routers) carry out a distributed computation to maximize the sum of individual source utilities. Based on this interpretation, it is shown in [15], [16] that queueing delay conveys the right congestion information and enables this distributed computation to maximize the aggregate utility under the right network conditions. [1] FAST TCP is based on this framework. The design of FAST TCP is realized on two levels. On the flow level [17], the window update algorithm is synthesized and analyzed to achieve the desired equilibrium properties such as good throughput and fairness as well as dynamic properties such as stability and responsiveness. On the packet level, FAST is implemented, taking into account of practical constraints, to best approximate the flow-level algorithm.

A. Window Update in FAST TCP

The core of FAST TCP is the window update algorithm, which determines the right congestion window size based on the current estimation of queueing delay. The window update function in FAST TCP can be described by the following flow-level algorithm.

$$\dot{w}(t) \quad = \quad \kappa(t) \cdot \left(1 - \frac{p(t)}{u(t)}\right) \tag{1}$$

where $\kappa(t) := \kappa(w(t), T(t))$ and $u(t) := u(w(t), T(t))$, where $w(t)$ is the congestion window, $p(t)$ is the end-to-end congestion feedback, $u(t)$ is the marginal utility function, and $T(t)$ is the Round-Trip Time (RTT), all at time t.

There are three design decisions in the window update algorithm:

- $\kappa(w, T)$: the choice of the gain function κ determines the dynamic properties such as stability and responsiveness, but does not affect the equilibrium properties. FAST uses $\kappa = \gamma\alpha$, where γ is set to .5 in our current algorithm, and α is the number of packets queued in a network in equilibrium belonging to each FAST TCP connection.
- $u(w, T)$: the choice of the marginal utility function u mainly determines equilibrium properties such as the equilibrium rate allocation and fairness. The marginal utility function in FAST TCP is α/x with x approximated as $x = w/T$.
- p: FAST uses queueing delay as p, whose dynamics is determined at links.

The packet-level implementation of the window update algorithm is slightly different because people generally believe that window increase should be no more aggressive than slow start, which at most doubles the window size in one RTT. FAST TCP periodically updates the TCP window based on the queueing delay `qdelay = avgRTT - baseRTT` (`baseRTT` is the propagation delay, and `avgRTT` comes from exponentially averaging instantaneous RTT samples) according to:

$$\texttt{w} \longleftarrow \min\left\{2\texttt{w},\ (1-\gamma)\texttt{w} + \gamma\left(\frac{\texttt{baseRTT}}{\texttt{avgRTT}}\texttt{w} + \alpha(\texttt{w},\ \texttt{qdelay})\right)\right\} \tag{2}$$

In our current prototype, we choose the function $\alpha(\texttt{w}, \texttt{qdelay})$ to be $\alpha(\texttt{w}, \texttt{qdelay}) = a\texttt{w}$, when `qdelay` is zero. In this case, FAST performs multiplicative increase and grows exponentially at rate a to a neighborhood of `qdelay` > 0. Then $\alpha(\texttt{w}, \texttt{qdelay})$ switches to a constant α. The constant α is the number of packets each flow attempts to maintain in the network buffer(s) at equilibrium. Currently, we have not modified the loss recovery mechanism in TCP and deactivate the window update function during loss recovery, and re-activate it once TCP exits recovery.

B. FAST Component Design

FAST TCP includes four independent components as shown in Figure 1. This independence allows individual components to be designed separately and upgraded asynchronously. The *data control* component determines *which* packets to transmit. This decision is important during loss recovery because of the need to infer queueing delay in the future when congestion window will be updated. We must maintain a steady stream of acks (with selective acknowledgment) from new transmissions in order to obtain unambiguous RTT measurements. *Window control* determines *how many* packets to transmit in each RTT and is responsible for congestion control. *Burstiness control* determines *when* to transmit packets as arriving acks free up space in the congestion window to smooth out the transmission rate. Whenever there is packet queueing on the reverse path, ack loss, or temporary CPU overload on the end hosts, *burstiness control* would take effect to regulate the instantaneous transmission of packets. These decisions are made based on information provided by the *estimation* component. *Window control* regulates packet transmission at a constant interval, while *burstiness control*, *data control*, and *estimation* work at a smaller timescale.

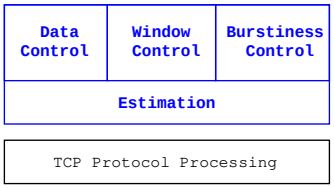

Fig. 1. FAST TCP architecture.

TCP is typically implemented as part of an OS kernel. Thus, for portability reason, we have divided the implementation of FAST TCP into an OS-independent module and an OS-dependent module that is tailored to each OS [18]. This design carries the maximum flexibility in that algorithmic changes will be exclusively on the OS-independent module, and only the OS-dependent module needs to be modified when porting FAST TCP to a new OS, or a newer version of an existing OS. We have found this division to be extremely convenient during our software development.

III. Experiment Networks

Today, scientists are mostly interested in being able to quickly transfer large data files among remote sites. Unfortunately, we don't yet understand or even recognize all the relevant issues in disk-to-disk transfer of large files. Thus, in our current set of experiments, we focused on the TCP

[1] By the right network conditions, we mean those conditions that are necessary for the existence of an equilibrium, for example, suffi cient buffering inside the network and FIFO AQM.

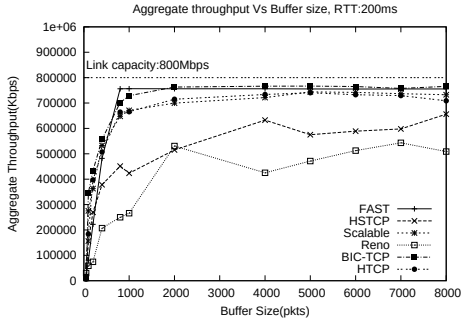

Fig. 2. Average `iperf` throughput vs. buffer size.

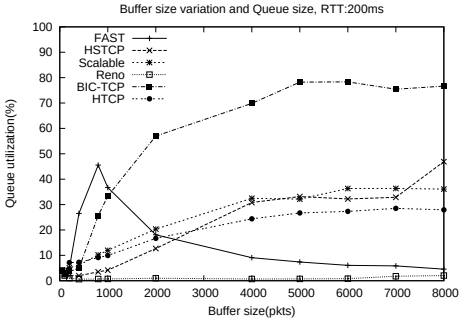

Fig. 3. Average queue occupancy (percentage of buffer space used) vs. buffer size.

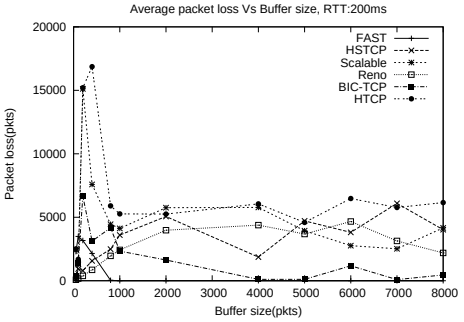

Fig. 4. The total number of lost packets vs. buffer size.

protocol behaviors in memory-to-memory data transfer using `iperf` [19].

We conducted several experiments on two real networks: Haystack's test network and TeraGrid. We also use our Dummynet test setup to replicate real network scenarios to verify our observations. We will give a detailed description of each network setup in this section.

A. Dummynet Setup and Basic Experiments

We realized during our earlier experiments that it was extremely difficult to understand protocol behaviors without examining in detail what happened within a network. Although there is a very successful emulated research network called Emulab [20], it does not provide very high-speed bottleneck links necessary for high-speed TCP testing. We constructed an emulated single-bottleneck high-speed testbed also capable of emulating large latencies.

Our current testbed consists of a sender, a receiver, and a host emulating different capacities and latencies. The sender and receiver are identical dual Intel Xeons of 2.66 GHz with 2 GB of system memory. Each machine is equipped with onboard dual Intel gigabit Ethernet controllers. The emulated testbed router runs Dummynet under FreeBSD 5.1. It has dual Intel Xeons of 3.06 GHz with 2 GB of system memory, onboard dual Intel gigabit Ethernet controllers. The Dummynet on the router is configured to have a capacity of 800 Mbps with FIFO and drop-tail queueing and varying path latencies depending on TCP port number. In order to see inside the emulated network, we have instrumented the network stacks at the TCP layer for both the sender and the receiver, and at the IP layer for the Dummynet router, to capture various state variables (see [6] for the detailed setup).

Our test setup is useful in giving insights on protocol behaviors, especially those inside the network, which are normally not visible to end-users. The Dummynet setup is an idealized network in the sense that we are able to control all aspects of system configuration, such as bandwidth, delay, cross traffic, and packet loss. It's a very useful tool in understanding and confirming protocol behaviors based on theoretical analysis. We have tested five implementations of high-speed TCPs including BIC-TCP [7], FAST [6], H-TCP [8], Highspeed TCP [4], and Scalable TCP [5], plus the standard Reno TCP, all in Linux kernel. BIC-TCP and FAST are based on the 2.4.22 kernel. H-TCP is based on the 2.4.23 kernel, and the others are based on the 2.4.19 kernel. There is very little difference in the base TCP networking code among the different versions, and we don't enable NAPI for 2.4.2x versions for Intel's gigabit Ethernet driver `e1000`. We increase the Linux `txqueuelen` to 10000 to prevent Linux from entering the `CWR` state for loss-based protocols. We also increase the Rx ring buffer size to 4096 for the `e1000` driver on both the sender and the receiver to help accommodate large bursts of packets.

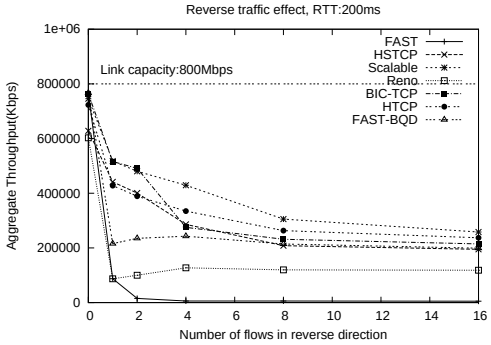

Fig. 5. Dummynet reverse congestion experiment.

For each protocol, we use `iperf` to generate two TCP streams going over a simulated 200 ms path with a bottleneck capacity of 800 Mbps. We vary the buffer size in the bottleneck link from 50 to 8000 packets. We show the aggregate `iperf` throughput, the queue occupancy, and the total number of packets lost at the bottleneck in Figures 2, 3, and 4, respectively. All protocols experience very low throughput when the buffer size is 50 packets. As buffer size increases, all protocols are able to improve the aggregate throughput. Unless the window reduction factor after each loss is extremely small as in the case of Scalable TCP, loss-based protocols generally need a reasonably large queue such that when TCP congestion window is reduced due to queue overflow, the resulting window is still large enough to sustain a high throughput. For n FAST TCP flows, the buffer size needs to be $n\alpha$ with each injecting α packets into the queue in equilibrium. For example, if α is fixed at 200 packets, two FAST streams would not reach equilibrium and stabilize if the bottleneck buffer size is less than 400 packets. Clearly, there is a need to scale α dynamically with the number of FAST TCP flows and bottleneck capacity, In our current implementation, we use a rudimentary algorithm to tune α based on achieved throughput.

Our instrumentation in Dummynet maintains a counter for the number of packets lost at the bottleneck link, and a moving average of the queue size inside Dummynet for data packets. We compute the queue occupancy as the arithmetic mean of the periodically sampled average queue sizes, normalized against the allocated buffer size. We compare queue occupancy and the number of packet losses in Figures 3 and 4.

As expected, all loss-based protocols increase the queue occupancy as the buffer size at the bottleneck link increases, since they have to force the queue to overflow in order to control the TCP congestion window. In contrast, FAST maintains a steady average queue, around the equilibrium as predicted in theory. Having large buffer sizes cannot eliminate packet losses for loss-based protocols, thus with a large buffer size, the network experiences both a high queue occupancy as well as relatively high packet losses for most loss-based protocols. In case of FAST TCP, once the buffer size is sufficiently large to support an equilibrium, there is a very stable queue with zero packet loss at the bottleneck.

For the loss-based protocols, we generally see a trade off between good throughput and low queue occupancy, in that good throughput generally comes at the cost of a very high average queue occupancy as in the case of BIC-TCP. This is consistent with how loss-based protocols operate, the better a protocol is able to maintain a large queue, the more likely the bottleneck link is fully utilized. This trade-off is not entirely obvious if one only examines `iperf` throughput without detailed understanding of how protocols operate.

Despite its many desirable properties, FAST TCP also faces practical challenges in real networks that require sound solutions. Some recent studies [21], [22] have shown that delay-based TCP protocols, including FAST, would have throughput problems if network congestion occurs in the reverse direction. This is a result of using end-to-end queueing delay as the congestion signal, in that queueing delay in the reverse direction cannot be distinguished from queueing delay in the forward direction. We are currently working on a solution that will detect the direction in which queueing delay occurs. We demonstrate the promise of this solution using a Dummynet experiment and compared it to the results of other TCP protocols. In this scenario, we use a number of TCP Reno streams in the reverse direction to congest the reverse path. This builds up a queue in the reverse path and slows down the returning ack traffic.

Reverse traffic builds up queueing delay on the reverse path, which increases the likelihood of ack loss and burstiness. In Figure 5, our experiment result shows that all protocols suffer performance degradation in the presence of congestion in the reverse path. The standard FAST implementation reduces its throughput considerably, consistent with previous work. Our current version of FAST with queueing delay detection(FAST-BQD), as shown in figure 5, was able to regain a significant portion of the lost throughput to be comparable to other protocols.

B. Haystack Experiments

The Radio Astronomy community has a great need for efficient high-bandwidth WANs to facilitate the collection of experimental data in near-real-time or real-time and to assist with collaborative efforts around the world. One of the challenges facing Radio Astronomers is to stream data at rates of up to 1 Gbps from multiple radio telescopes to a correlation center in real-time or near-real-time. In order to be able to achieve this, high capacity networks and highly efficient network protocols are required. In this section, we describe the experimental setup for data transfers using advanced TCP protocols between Japan and the United States. These transfers occur across the high speed Asia Pacific Advanced Network/TransPac and across Internet2's 10 Gbps Abilene network. These networks are regularly used by Massachusetts Institute of Technology (MIT) Haystack Observatory to transfer data from Japan as part of its International e-VLBI program.

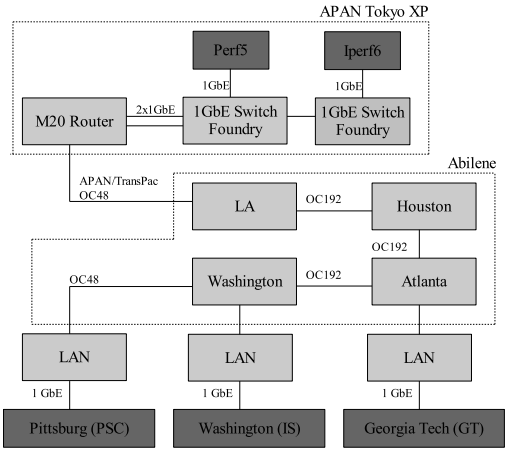

Fig. 6. Haystack topology.

MIT Haystack Observatory is an interdisciplinary research center engaged in radio astronomy, geodesy, and atmospheric sciences [23]. One application that is taking advantage of high speed Research and Education networks is the Very Long Baseline Interferometry (VLBI) project. VLBI is a radio astronomy technique that combines multiple radio telescopes around the world into a single coherent instrument equivalent in resolving power to a radio telescope the same size as the physical separation of the telescopes. Traditionally, VLBI data is transferred from stations to a central correlation center(where it is processed) by recording the data onto magnetic media(such as magnetic tape/disc) and then physically shipping the data to the correlation center. Electronic-VLBI (e-VLBI) is able to make use of high speed research and education networks to electronically transmit from the telescopes to the correlation center. e-VLBI currently supports production level transfer of data between radio telescopes in Kashima (managed by the National Institute of Communications and Information Technology) and Tsukuba (managed by the Geographical Survey Institute) and Washington, where data is either transferred or correlated at the United States Naval Observatory (USNO). All of these transfers make use of the Asia Pacific Advanced Network/Transpac link between Japan and the United States. Data sets for these transfers are typically of the order of 500 GB per station per month. Current transfers use TCP stack tuning and tools such as BBFTP [24] to try to improve the transfer performance over round trip times that are typically between 170–200 ms, in a manner that is TCP-friendly. This solution is temporarily, awaiting more permanent solutions in the form of advanced TCP protocol stacks. The use of advanced TCP protocol stacks to achieve similar, or better performance would allow radio astronomers to use existing tools to transfer their data. In this section, we describe some tests performed between the Tokyo XP site in Japan and a site in Washington that is regularly used for e-VLBI transfers. These tests serve to highlight the complex issues in the development and deployment of new TCP protocols and provide a good starting point for evaluating the efficiency of these new protocols.

The experiments consisted of a number of data transfers over the Pacific link between US and Japan using the Reno-TCP, HSTCP, UDP and FAST-TCP protocols. In the US, computers in Pittsburgh (PS), Georgia Tech (GT) and Washington at the ISI institute (IS) are employed to serve as senders and receivers to and from the Tokyo server (Perf5). The network topology between these sites is depicted in Figure 6. The Perf5

is an Intel PIII 1.4 GHz with 1 GB of memory, IS is a PIII 1.2 GHz machine with 256MB of memory, GT is a 2.66 GHz Intel Xeon with 2 GB of memory, and PS is a dual Xeon 3 GHz with 2 GB of memory.

C. TeraGrid Experiments

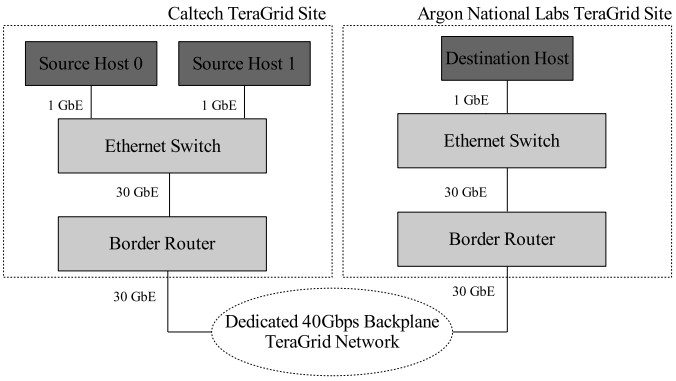

Fig. 7. TeraGrid topology.

TeraGrid [25] is a Grid computing initiative with the aim of becoming the world's largest distributed computing infrastructure for open scientific research. When complete, it will feature 20 teraflops of computing power and 1 petabyte of data storage connected, with 5 sites connected over a 30 Gbps network [25]. It has over 700 users in fields including astronomy, biomedical research and weather forecasting. The five sites of TeraGrid have different specializations; the Caltech site performs data collection analysis, Argonne National Labs (ANL) performs data visualization, the National Center for Supercomputing Applications (NCSA) has facilities for processing intensive applications and the San Diego Supercomputing Center at UCSD (SDSC) specializes in data-intensive applications. The design of the TeraGrid, with 30 Gbps of capacity to each site, facilitates the sharing of large datasets across the different sites. Efficient protocols to make use of this capacity are essential.

The experiments consist of a number of transfers from two hosts at Caltech to a receiving host at ANL. The bottleneck capacity between Caltech and ANL is 1 Gbps with an RTT of 55 ms. The network architecture is shown in Figure 7. The Caltech computers are dual-processor 1.3 GHz Itanium2 machines with 6 GB of main memory, running SuSE SLES 8.0. The ANL receiver is a dual-processor 1.3 GHz Itanium2 machine with 4 GB of memory running SuSE SLES 8.0 GOLD. Due to an unknown incompatibility between 1500-byte MTU and NFS, all of our experiments used jumbo frames of 9000 bytes.

IV. Performance Evaluation

There have been on going studies to evaluate the current set of high-speed TCP protocols to determine the best overall protocol. Many insights have been revealed by some of these studies [26]. Our focus is not to determine a winner since many of the protocols are under intensive and rapid improvement. Rather, we hope to achieve in-depth understanding of the environments in which various protocols would perform well, and the causes of poor performance.

Ideally, we would look at various statistics in the network, such as bandwidth utilization, queueing delay, and packet loss, as well as host statistics such as `iperf` throughput, CPU utilization, and in the case of FAST TCP, many internal Linux kernel variables. However, it is very difficult to collect statistics on the performance of the networks themselves, even in research networks such as TeraGrid due to the tremendous effort required to collect them. We have to rely on end-to-end mechanisms to obtain as much information as we can. We use a short UDP experiment prior to each TCP experiment to gauge the packet loss rate, and we use `ping` measurement every ten seconds to sample the queueing delay in the network.

Figures 8 and 9 show the statistics that we collected for one experiment where a single FAST TCP was used to transfer data from an Internet2 machine in Atlanta to a Haystack machine in Japan. Figure 8 shows the `iperf` throughput over time, the CPU utilization, and the ping measurements. All three entities are fairly stable over time, as expected. Furthermore, we are able to see from Figure 9 that the congestion window is constant after the initial ramp-up, which is consistent with our expectation that with no cross-traffic, FAST should fully utilize the 1 Gbps bottleneck link, while maintaining a constant window in equilibrium. Having this level of information helps one understand that the oscillation in `iperf` throughput is not due to the protocol or the network, but more likely due to the mechanism in transmitting packets inside the Linux kernel. We believe it is necessary to analyze and understand all of the statistics shown in figures like these two in order to draw the right conclusions about TCP protocol behaviors.

A. Summary of Results

We observe that TCP Reno would have stable throughput with little packet loss only when transmission rate is relatively small compared to the bottleneck, which may be either inside the network or the host Gigabit Ethernet card. However, when packet transmission approaches the line speed, packet loss happens more frequently, and throughput oscillates severely. Many causes of packet loss are unfortunately not known to us. We noticed that a lot of times, sending a UDP stream with a constant rate revealed unexpected hidden capacity bottlenecks on the path that is much smaller than the advertised bottleneck link capacity. We believe there is an imperative need for better instrumentation of network elements and the understanding of the limitations of various network elements.

The protocol implementation also plays a very critical role in determining the overall performance. Specifically, we found that the TCP loss recovery in Linux can be very inefficient by often taking an excessively long period of time (tens and even hundreds of seconds) to recover from packet loss. We provide some understanding and are currently working on a solutions to resolve this performance bottleneck. In addition, we discover an implementation bug with the way Linux increases the receiver's advertised window.

B. Host (Linux) Issues

We are only showing two major issues that we have noticed in our testing so far, and we are likely to see other host related issues that can affect high-speed data transfers using TCP. In order to discover and understand these issues, it is absolutely critical to export as many internal kernel variables to the user space as web100 [27] is already doing. We would like to emphasize that lots of effort should go into understanding and resolving various host related issues before any serious deployment of a high-speed TCP protocol.

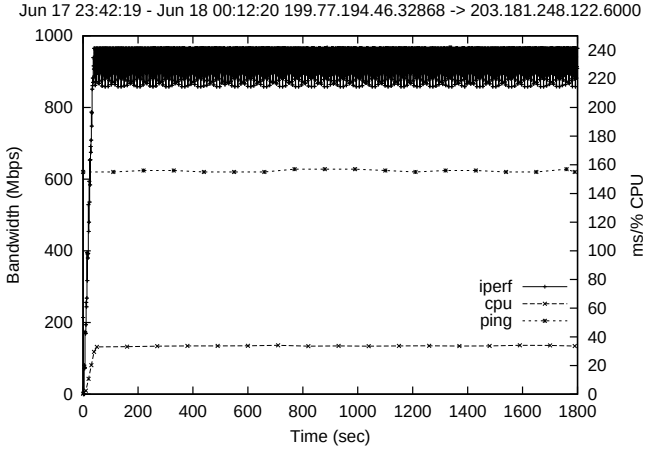

Fig. 8. `iperf`, CPU, and `ping` data of a single FAST flow with $\alpha = 100$.

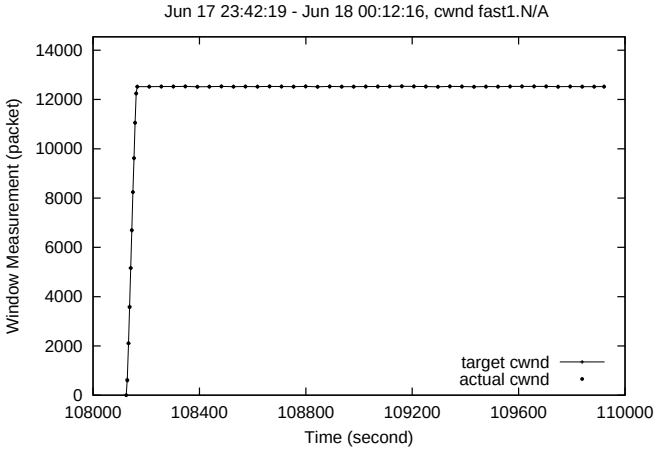

Fig. 9. Congestion window evolution over time.

1) Loss Recovery: We frequently encountered extremely slow recovery after packet loss events. This is consistent with observation made and documented by other developers of high-speed TCP protocols. In fact, the developers of both H-TCP and Scalable TCP have released patches [5], [8] to improve the loss recovery behavior of the standard Linux implementation. The patches generally include two parts. The first part speeds up SACK processing, and the second part regulates the congestion window so it stays at some reasonable value during loss recovery. In fact, congestion window often becomes one segment in many of our experiments. Since the slow loss recovery did significantly reduce TCP throughput in some of the experiments using real networks, we decide to take a closer look at the cause of poor loss recovery.

Figure 10 shows an experiment from a machine at Internet2's Atlanta exchange point to a machine belonging to the Haystack project in Japan. The first FAST stream was started at time zero, and the second stream was started 300 seconds later on the same machine and terminated after 600 seconds. As expected, the first flow was able to fully utilize the bottleneck capacity of 1 Gbps at the gigabit Ethernet card in a reasonably stable manner. When both flows were sharing the gigabit Ethernet card, each obtained a stable and fair share of the bandwidth. Both the `ping` and CPU utilization graphs are stable over time. However, we observe that the second FAST flow suffered a long period of silence starting around time 600, and lasted for nearly 100 seconds.

Under normal operations, Linux maintains two variables `snd_cwnd` for the current congestion window, and `packets_out` for the number of unacknowledged packets. A TCP sender can only transmit packets up to the difference between `snd_cwnd` and `packets_out` so that the total estimated packets in transit does not exceed `snd_cwnd`.

During loss recovery, packet accounting becomes more complicated. Linux limits `snd_cwnd` to the estimated number of packets in transit in the network. This estimate is the sum of `packets_out` and `retrans_out`(the number of retransmissions in flight) minus the number of packets estimated to have left the network, `left_out`, where `left_out` is the sum of `sacked_out` (the number of non-duplicate SACKs received) and `lost_out` (the number of lost packets). `snd_cwnd` is capped by this difference plus some threshold values during loss recovery in various places in the kernel code.

Figure 11 gives an enlarged view of what was happening in terms of the aforementioned Linux's internal TCP state variables. We observed that there was a time-out early during the recovery period, and all outstanding packets that were not SACKed were marked as lost at the time. Of course, some of these "lost" packets were later received as SACKs. As shown in the figure, `sacked_out` increased with time and then stayed constant as all outstanding SACKs were received, while `lost_out` decreased sharply in the beginning as "lost" packets were remarked as SACKs, and then decreased slowly as lost packets were retransmitted.

There are a few problems with this scenario. First, after a time-out, TCP must set the congestion window `snd_cwnd` to one. However, in the Linux TCP implementation, `snd_cwnd` would never increase during loss recovery (see functions `tcp_ack` and `tcp_may_raise_window` in `tcp_input.c` in the kernel source code). In this particular example, 711 packets were lost, and since `snd_cwnd` was capped at one, it took 711 RTTs to complete loss recovery. The RTT of the path is around 160 ms so the loss recovery took a total of around 113 seconds of zero goodput! Second, the logic behind capping `snd_cwnd` near the number of packets in flight is the following. In the steady state (the exact details depend on the definition for a particular TCP variant), in one RTT, a TCP sender should not send more than the number of acknowledged packets from the previous RTT. In this case, `snd_cwnd` is an indication of the current network congestion as it is intended to. However, during loss recovery, the number of packets in flight isn't computed per RTT in the Linux implementation. In fact, there is one global variable for packets in flight (the actual implementation uses the value of `packets_out + retrans_out - sacked_out - lost_out`) being updated throughout the loss recovery period, which often spans tens and even hundreds of RTTs. Thus, the number of packets "in flight" is no longer an accurate measure of network congestion, and capping `snd_cwnd` around this value is not correct. In FAST implementation, we are currently working on a scheme that would allow normal window update even during loss recovery so we can set a correct `snd_cwnd` based on some conservative estimate of network congestion.

2) MTU Issue with Linux Receivers: We found in quite a few experiments that there was a mysterious receiver window limitation of 3,147,776 bytes, which was confirmed by using a web100 kernel at the receiver in some of these instances. Fur-

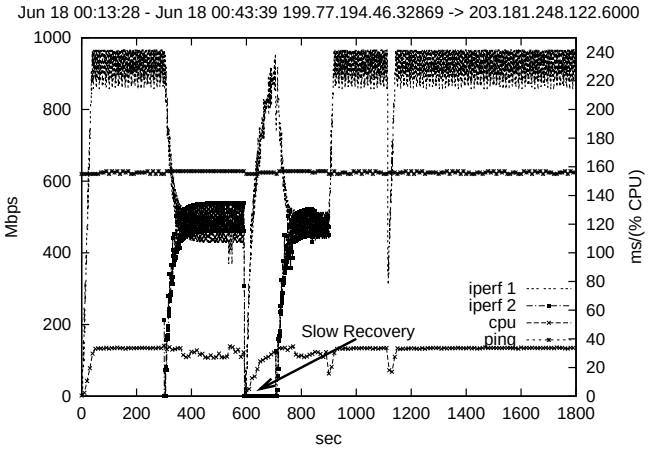

Fig. 10. `iperf`, CPU, and `ping` data of a dynamic scenario with two FAST flows.

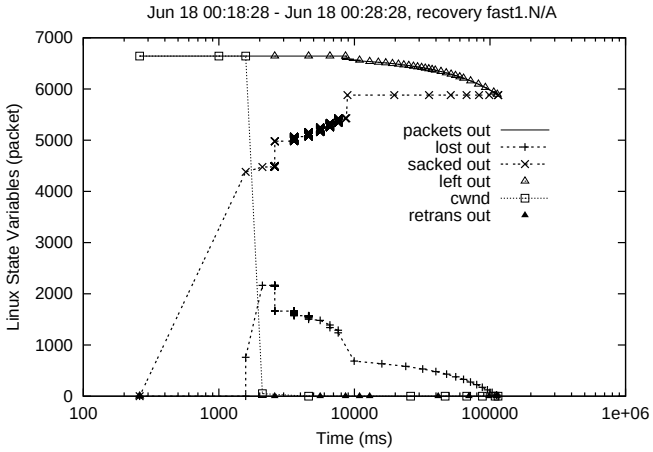

Fig. 11. Detailed view of default Linux loss recovery in Figure 10 at time 600sec. Note a lost packet is first at time 0, and after around 1000 ms a timeout occurs. At first cwnd is over 6000 and drops to and remains at 1 after the timeout.

thermore, with the same receiver, TCP flows from a different sender would not see this limitation. It turns out that the only obvious configuration difference between the two senders is the setting of the MTU on the network interface card.

This behavior is quite unexpected because MTU is a data link layer concept that should not affect the transport layer performance. In order to narrow down the problem, we repeat the experiment in our Dummynet setup. It turns out that as long as both the receiver and the sender have the same MTU, we get the expected performance of close to 800 Mbps. When the sender has a 1500-byte MTU, while the receiver has a 9000-byte MTU, we see the receiver limitation, which may constrain the throughput depending on the RTT. When the situation is reversed between the sender and receiver, we can again obtain close to 800 Mbps of throughput. It appears that the MTU issue only affects the receiver, but not the sender.

Upon instrumenting the receiver on our Dummynet, we realize that the 9000-byte MTU is not the cause for the problem. Linux kernel 2.4.22 has a bug in that certain combination of parameters would cause the receiver's advertised window not to increase beyond a certain value. Specifically, with an `iperf` server requesting 64 MB of memory, 9000-byte MTU on the receiver side, and 1500-byte MTU on the sender side, the TCP receiver window will stop increasing beyond 3,148,472 bytes. Since the advertised window has to be in multiple of Maximum Segment Size (MSS) of 1,448 bytes, the receiver window of 3,148,472 bytes translates into 3,147,952 bytes through integer division and then multiplication by 1,448. Furthermore, because of window scaling, the number of bytes needs to be divided by 1,024 at the receiver, and then multiplied again at the sender to give a receiver advertised window of 3,147,776 bytes. Because of the relatively small receiver's window size, this limitation is likely to affect most connections going over high-bandwidth and long-latency paths as it did in our case.

C. Network Issues

Performance problems within networks are the most difficult to debug, precisely because of our inability to access the internal state of the network, and even if we do have the ability, the lack of tools to effectively analyze the massive amount of real-time data. The best solution would be for network operators, equipment manufactures, protocol researchers, and OS implementers to educate each other about the requirement of various components to build a more efficient data networking infrastructure (both hardware and software) that is suitable for Grid applications. Furthermore, it is necessary to have end-to-end performance monitoring tools such as web100 to isolate possible source(s) of poor performance.

1) Interaction between Networks and Protocols: Ideally, TCP protocols should be evaluated based on their performance that results from the algorithm design, but not from other external factors. One clear message from our experiments is that a wide range of issues impact the performance of TCP protocols, and it is imperative to look at the details to separate the performance issues of the protocol from the those with the network, operating systems, or applications. Thus, to fairly evaluate the performance of a TCP protocol, it is useful to first understand its performance in a controlled environment.

In a controlled environment such as Dummynet, it is possible to reconcile the protocol's theoretical performance to experiment results, and be sure that there are no mysterious bottlenecks and issues affecting the result. Only after the protocols behaviors are consistent with theory in a controlled environment, it is reasonable to reconcile the behaviors of the protocol in a production environment where many more external variables exist.

The experiments on TeraGrid serve to illustrate this message. Two sender hosts at Caltech were used to perform a number of transfers to ANL using both FAST and RENO. Flow 1 was started at time 0 from source 1, and after 600 seconds, flow 2 from source 1 was started as well as flow 3 from source 2. At 1200-second mark, flow 4 coming from source 2 was started. At 1800-second mark, flow 2 was stopped, and at 2400-second mark, flow 1 was stopped.

Dummynet was used to emulate the TeraGrid experiment to verify the performance of FAST and RENO in a controlled network environment with similar characteristics. Due to the processor limits, the Dummynet bottleneck link was set to 800 Mbps as compared to the 1 Gbps TeraGrid capacity, and propagation delay was set to 55 ms. The TeraGrid experimental results for FAST and RENO are plotted in Figures 15 and 14. Figures 13 and 12 show the FAST and RENO throughput using the Dummynet testbed.

The Dummynet results are consistent with what theory predicts about the behaviors, that is, the bandwidth is fully utilized, the capacity is shared fairly among the flows, and

the flow rates are stable over time. However, there are two main differences between the Dummynet and TeraGrid results: the fairness among the flows, and the maximum throughput achieved by the aggregate. The Dummynet results guide us to understand what the protocol is capable of, and we must now try to reconcile why the performance differences arise in the production environment. There are a number of factors which may contribute to the differences.

The difference in throughput seems to be an issue with the network or host bottleneck. Before the experiment began, to probe the capacity of the path, a 1 Gbps UDP stream from each source was started. A high loss rate was observed, and the UDP stream from source 1 achieved 348 Mbps, and source 2 achieved 335 Mbps, indicating a path capacity of 683 Mbps as compared to the 1 Gbps expected from the network. Given this bottleneck capacity at the start of the experiment, FAST seemed to have utilized it fully. The nature of this bottleneck remains unknown, however the UDP test shows that the bottleneck is the network/host rather than the protocol.

Note that between the 1900-second and 2000-second marks, both the RENO and FAST throughput decreased in the TeraGrid experiments. This was caused by background traffic, a UDP flow inadvertently started by the test script to probe the network capacity.

The difference in flow fairness can be attributed to the behavior of the protocol given different samples of $baseRTT$. The $baseRTT$ sampled by four flows in the TeraGrid experiment are 57.63, 54.13, 54.16, and 54.11 ms, whereas the flows in Dummynet sampled 51.28, 51.67, 51.48, and 51.57 ms. Slight differences in flow start times and background traffic result in different $baseRTT$ samples, so it is expected that in different environments, the fairness between flows has some variation. Also note that in the Dummynet experiments, all four sources actually originated from the same host, whilst in the TeraGrid experiment two separate senders were used.

V. Conclusion

The research community has made significant progress in understanding the necessary requirements for a TCP implementation that is capable of high performance over high-capacity and long-latency networks. A number of very promising proposals have been made as the possible replacement to the current TCP Reno. While a lot of work has gone into analyzing various high-speed protocols, much more work needs to go into understanding the practical issues affecting TCP performance.

In this study, we look at the performance of mainly FAST TCP on two high-speed networks. We discover that FAST TCP often has very good performance, in line with those predicted by theory. However, there are also a number of scenarios where FAST doesn't perform as well as expected. We share some of our experiment methodologies that we believe are extremely useful in evaluating protocol performance and limited experiences in understanding the sources of poor performance.

A common practice in evaluating TCP protocols is to focus on just the throughput since that is what end-users see, which is inappropriate. For example, an aggressive TCP protocol may have very good throughput and possibly even stability, but probably will be extremely unfair to all the other competing traffic. It is difficult to claim superiority of one protocol over another if there is a trade-off between achievable throughput and impact on the background traffic, which is typically not seen by end-users. One must be very careful in getting an comprehensive set of data when evaluating protocol performance. In order for readers to make correct interpretation of results of real-network experiments, those doing the evaluations should include as much information about the network as possible along with the experiment results. For example, it is not really meaningful to simply provide an output of `iperf`, without stating the bottleneck link capacity, the amount of background traffic, and average loss rate. While it is difficult getting such information for all experiments, it is practically impossible to make correct interpretation of `iperf` throughput without associated network information.

Based on our experience with the current set of tests, much remains to be understood in issues that affect TCP performance in high-speed networks. Designing a good high-speed protocol in theory is only the first step to obtaining good performance on high-speed networks. The protocol research community as a whole needs to spend more time to understand the practical limitations that affect TCP performance and strive to overcome these bottlenecks in implementation. In addition, protocol researchers, OS implementers, network providers, and equipment manufactures need to educate each other about the issues each faces in high-speed data networking to reach a common and better understanding necessary for making TCP over high-speed networks a reality.

VI. Acknowledgments

We acknowledge the support of many individuals and organizations beyond the list of authors, who contributed their time and resources behind the scenes. In particular, for the Haystack experiments, we would like to thank Alan Whitney (Haystack), Dr. Masaki Hirabaru (National Institute of Information and Communications Technology NICT), Guy Almes and Stas Shalunov (Internet2), the staff at APAN/TransPAC for their assistance in testing, server and router configuration, as well as David Richardson (Pacific Northwest GigaPoP), Matt Mathis (Pittsburgh Supercomputing Center), Scott Friedric and Cas D'Angelo (Georgia Tech), John H. Moore (North Carolina State University), Tom Lehman (University of Southern California Information Sciences Institute USC ISI), and Raj Jayaram from Caltech.

For the TeraGrid experiments, we would like to thank Linda Winkler (Argonne National Lab), Julian Bunn, Yang Xia, James Pool, James Patton, Chip Chapman, and Suresh Singh from CACR Caltech.

This work is performed as part of the FAST Project supported by NSF, Caltech, ARO, AFOSR, and Cisco.

References

[1] IBM, "Ibm grid computing home page," URL: http://www.ibm.com/grid.

[2] University of Chicago, "The globus alliance," 2002 –, URL: http://www.globus.org.

[3] J. Unger and M. Haynos, "A visual tour of open grid services architecture," URL: http://www-106.ibm.com/developerworks/grid/library/gr-visual/.

[4] Sally Floyd, "HighSpeed TCP for large congestion windows," Internet draft draft-fbyd-tcp-highspeed-02.txt, work in progress, `http://www.icir.org/floyd/hstcp.html`, February 2003.

[5] Tom Kelly, "Scalable TCP: Improving performance in highspeed wide area networks," Submitted for publication, `http://www-lce.eng.cam.ac.uk/~ctk21/scalable/`, December 2002.

[6] Cheng Jin, David X. Wei, and Steven H. Low, "TCP FAST: motivation, architecture, algorithms, performance," in *Proceedings of IEEE Infocom*, March 2004, `http://netlab.caltech.edu`.

[7] L. Xu, K. Harfoush, and I. Rhee, "Binary increase congestion control for fast long-distance networks," in *Proc. of IEEE Infocom*, 2004.

[8] R. Shorten, D. Leith, J. Foy, and R. Kilduff, "Analysis and design of congestion control in synchronised communication networks," in *Proc. of 12th Yale Workshop on Adaptive and Learning Systems*, May 2003, `www.hamilton.ie/doug_leith.htm`.

[9] I. Foster, C. Kesselman, J. Nick, and S. Tuecke, "The physiology of the grid: An open grid service architecture for distributed systems integration," 2003, URL: http://www.globus.org/research/papers/ogsa.pdf.

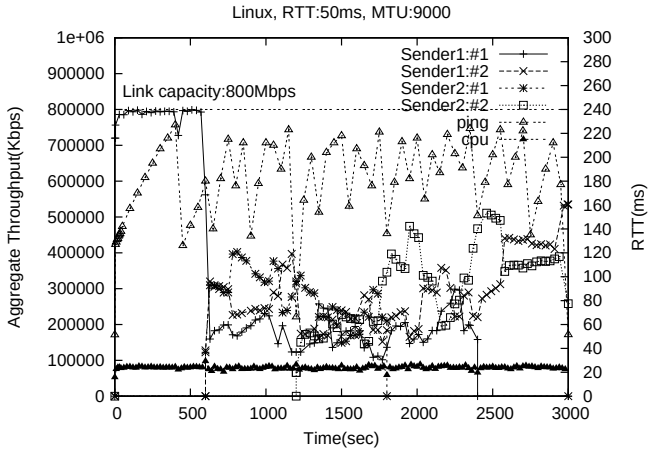

Fig. 12. Reno Dummynet throughput.

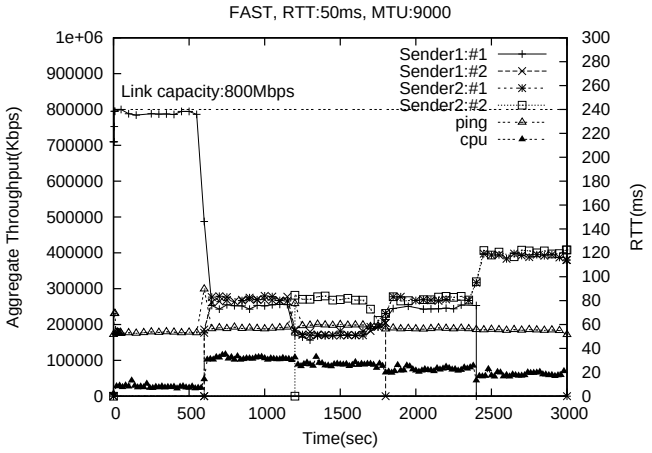

Fig. 13. FAST Dummynet throughput.

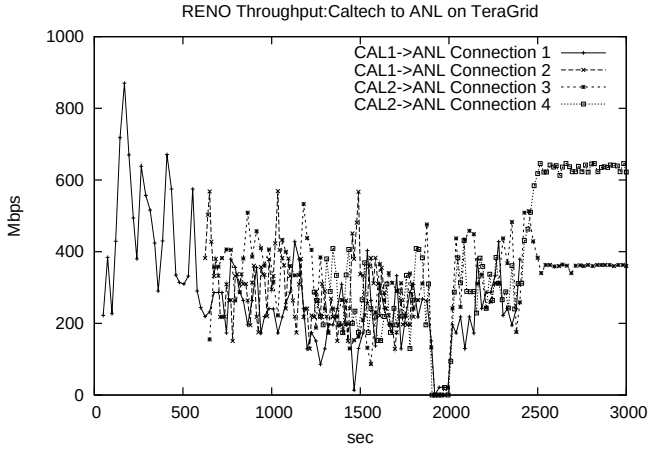

Fig. 14. Reno TeraGrid throughput

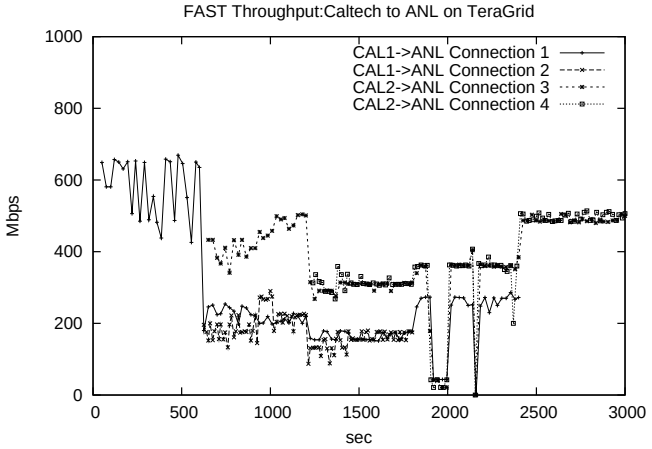

Fig. 15. FAST TeraGrid throughput.

[10] IETF, "Ip storage (ips) charter," URL: http://www.ietf.org/html.charters/ips-charter.html.

[11] Kevin Fall and Sally Floyd, "Simulations–based comparisons of Tahoe, Reno and SACK TCP," *ACM Computer Communication Review*, vol. 26, no. 3, pp. 5–21, July 1996, `ftp://ftp.ee.lbl.gov/papers/sacks.ps.Z`.

[12] S. Floyd and T. Henderson, "The NewReno modification to TCP's Fast Recovery algorithm," RFC 2582, `http://www.faqs.org/rfcs/rfc2582.html`, April 1999.

[13] Steven H. Low, Fernando Paganini, and John C. Doyle, "Internet congestion control," *IEEE Control Systems Magazine*, vol. 22, no. 1, pp. 28–43, February 2002.

[14] S. H. Low and R. Srikant, "A mathematical framework for designing a low-loss, low-delay internet," *Networks and Spatial Economics, special issue on "Crossovers between transportation planning and telecommunications", E. Altman and L. Wynter*, January-February 2003.

[15] Steven H. Low, Larry Peterson, and Limin Wang, "Understanding Vegas: a duality model," *J. of ACM*, vol. 49, no. 2, pp. 207–235, March 2002, `http://netlab.caltech.edu`.

[16] Steven H. Low, "A duality model of TCP and queue management algorithms," *IEEE/ACM Trans. on Networking*, vol. 11, no. 4, pp. 525–536, August 2003, `http://netlab.caltech.edu`.

[17] Cheng Jin, David X. Wei, and Steven H. Low, "The case for delay-based congestion control," in *Proc. of IEEE Computer Communication Workshop (CCW)*, October 2003.

[18] Netlab @Caltech, "Fast homepage," URL: http://netlab.caltech.edu/FAST.

[19] NLANR Distributed Applications Support Team, "Iperf – the tcp/udp bandwidth measurement tool," URL: http://dast.nlanr.net/Projects/Iperf/.

[20] University of Utah, "Emulab - network emulation testbed home," 2000 –, URL: http://www.emulab.net.

[21] L. A. Grieco and S. Mascolo, "Westwood+ tcp ns-2 simulation results: the effect of reverse traffic," URL: http://www-ictserv.poliba.it/mascolo/TCP%20Westwood/Reverse/reverse.htm.

[22] H. Bullot and L. Cottrell, "Tcp stacks testbed," URL: http://www.slac.stanford.edu/ hadrien/reversefast/index.html.

[23] Haystack Observatory, "General information," URL: http://web.haystack.mit.edu/haystack/general.html.

[24] Gilles Farrache, "bbftp," URL: http://doc.in2p3.fr/bbftp/.

[25] TeraGrid, "The teragrid project," 1997 –, URL: http://www.teragrid.org.

[26] H. Bullot and L. Cottrell, "Tcp stacks testbed," `http://www-iepm.slac.stanford.edu/bw/tcp-eval/`.

[27] Web100, ," `http://www.web100.org`, 2000.