# Testing FAST TCP over Abilene

Stanislav Shalunov ⟨shalunov@internet2.edu⟩

2nd PFLDnet Workshop, Argonne, February 2004

## Abstract

A 2.5 Gb/s production Abilene circuit to SoX GigaPoP is congested by FAST flows for 30 minutes using a novel test tool, without adverse effects on production or other test traffic. The topic of internal queuing in Linux is touched.

## 1 Introduction

FAST TCP [5] [2] is an advanced version of TCP congestion control that uses delay, in addition to loss, as its source of knowledge about the state of congestion at the bottleneck, and, therefore, has a potential for using the network bottleneck link capacity fully without causing the output queue on the bottleneck interface to grow significantly. The purposes of this experiment were:

1. to evaluate the safety of deploying FAST TCP, in its current version, on Abilene [1] in particular and on Internet2 networks in general,

2. to understand better the interaction of FAST TCP traffic with conventional traffic at a high-speed congested bottleneck,

3. to learn more about the behavior of the current FAST TCP implementation on a production network and about the degree to which such behavior agrees with the theoretical model, and

4. to demonstrate that high-speed (network-saturating) traffic flows that do not adversely affect production traffic to a significant extent are possible today without the need of any quality-of-service techniques or active queue management.

The tests appear to have contributed towards the fulfillment of these goals: FAST TCP, even if its behavior does not currently, due to implementation issues, agree completely with the theoretical model, appears to be a safe tool for obtaining high speeds for data transfers on production networks; one can further hope that as the implementation is improved, the tool will become even more efficient and will affect the queue depth at the bottleneck to an even lesser extent.

## 2 Test Setup and Configuration

A series of progressively longer tests was conducted with an increasing number of flows; the primary goal of the initial tests was to convince the network operators involved that the transport protocol possesses the requisite congestion avoidance properties and, therefore, a significant adverse effect on production traffic is unlikely; a secondary goal was to gain more experience with FAST TCP in general and the test configuration in particular so that extraordinary results could be recognized as such. Only the last experiment is described here. The previous experiments [7] [8] provided generally similar results.

The tests used eight machines, all connected via Gigabit Ethernet, to produce four TCP flows. Two flows were FAST TCP; they added enough traffic to saturate a bottleneck OC-48 link with a capacity of 2.5 Gb/s. One flow was conventional FreeBSD Reno TCP; this flow was CPU-limited at the sender and was used to evaluate the impact of FAST TCP flows on production traffic. One flow was conventional Linux Reno TCP; this flow was congestion-limited at its source (Gigabit Ethernet network interface card) and was used to further understand the interaction between FAST TCP and conventional TCP.

The machines were located at five sites throughout the continental United States:

**SoX** Southern Crossroads GigaPoP in Atlanta,

**PNW** Pacific Northwest GigaPoP in Seattle,

**PSC** Pittsburgh Supercomputing Center in Pittsburgh,

**NC-ITEC** North Carolina Internet2 Technology Evaluation Center in Raleigh, and

**STTL** The measurement rack in a Qwest PoP next to the Abilene core router in Seattle.

The eight machines were as follows:

**fast1** a Linux 2.4.20 machine with FAST TCP kernel patches, at SoX,

**fast2** a Linux 2.4.20 machine with FAST TCP kernel patches, at SoX,

**fast3** a Linux 2.4.20 machine with conventional TCP, at SoX,

**fast4** a Linux 2.4.20 machine with FAST TCP kernel patches, at NC-ITEC,

**fast5** a Linux 2.4.20 machine with FAST TCP kernel patches, at PNW,

**fast6** a Linux 2.4.20 machine with conventional TCP, at PSC,

**gigatcp1** a FreeBSD 4.3-RELEASE machine with conventional TCP, at SoX, and

**nms1-sttl** a FreeBSD 4.6-RELEASE machine with conventional TCP, at STTL.

The machines fast3 and fast6 ran Linux on hardware identical to that of the other machines with "fast" in their names; FAST TCP patches were installed on these machines, but were not active during the test and thus conventional Linux TCP stack was used. The fast1, fast2, and fast3 machines at SoX were connected to different multimode fiber Gigabit Ethernet ports on a Juniper M-40 router with a 10 Gb/s connection to the SoX border router (a Juniper M-40 as well). The gigatcp1 machine was connected to a Summit 7i switch with a connection to the SoX border router. The fast4 machine was, at the time of the test, connected to the NC-ITEC border router (Cisco GSR) with an Engine 0 Gigabit Ethernet card (it was later moved behind a Gigabit Ethernet switch). The fast5 machine is connected, via PNW Gigabit Ethernet infrastructure, to the two PNW border routers. The nms1-sttl machine has a direct multimode Gigabit Ethernet connection to the Abilene core router, a Juniper T-640, in Seattle. The hardware specification of the FAST machines is described in Fig. 1; for gigatcp1, this information can be found in Fig. 2; for nms1-sttl, on Fig. 3.

| |
|---|
| Supermicro SYS-6013P-8 barebone system |
| Motherboard X5DPR-8G2; Intel E7501 chipset |
| Dual 2.66 GHz Xeon with FSB at 522 MHz |
| 2 GB of DDR Kingston ECC memory |
| 1 64-bit, 133 MHz PCI-X (full length) |
| 1 64-bit, 66 MHz PCI (low profile) |
| Intel 82546EB 2-port Gigabit Ethernet card |
| Adaptec AIC-7902 dual channel Ultra320 SCSI |
| 2015S Zero-channel 36 GB IBM U160 Hard disk |
| 1 U Rackmount with 400 W power supply |
| SysKonnect Gigabit Ethernet (multi-mode, 1-port) |

Figure 1: Hardware specification of FAST TCP machines.

| |
|---|
| SuperMicro 370DE6 Motherboard |
| Two 256 MB ECC registered DIMMs |
| Two IBM 18.2 GB Ultra160 drives |
| Two Intel Pentium III/1000 (133 MHz FSB) |
| SuperMicro SC760 Chassis |
| 3Com 3c985B-SX Gigabit Ethernet |

Figure 2: Hardware specification of the gigatcp1 machine.

The four paths for test flows were as follows:

PATH1 nms1-sttl $\rightarrow$ gigatcp1 (stock FreeBSD Reno),

Intel SCB2 motherboard; 512 kB L2 cache
Two Pentium III 1.263 GHz; 133 MHz FSB
Two 512 MB ECC registered RAM
Two Seagate 18 GB SCSI (ST318406LC)
SysConnect Gigabit Ethernet SK-9843 SX

Figure 3: Hardware specification of the nms1-sttl machine.

PATH2 fast4 → fast1 (Linux with FAST patches),

PATH3 fast5 → fast2 (Linux with FAST patches),

PATH4 fast6 → fast3 (stock Linux Reno).

If these paths are considered individually, each has a bottleneck of 1 Gb/s (in the sender network interface card, with an identical bottleneck before the receiver network interface card). Each of these paths passes through the OC-48 link from the Abilene core node in Atlanta (ATLA) to the SoX border router. The flows are thus capable of saturating that link with test traffic. Behind that link are the networks of approximately 40 institutions [9].

## 3 Test Tool Description

A custom tool was used to conduct measurements of throughput and delay.

### 3.1 Rationale for a Custom Tool

Since FAST TCP to a significant extent relies on round-trip delay measurements to determine its congestion window and, thus, sending rate, it is desirable to measure both throughput and delay during the tests concurrently.

One approach to measuring delay would be to use a tool such as PING to send ICMP messages to measure the RTT; the disadvantages of such approach would include:

1. poor suitability of PING for sending high-rate test flows, and therefore, lower quantity of data,
2. the necessity to correlate delay data obtained with PING with throughput data in the face of PING typically sending packets a little slower (by an amount that depends on load) than requested,
3. ICMP packets receiving different treatment in the hosts, and
4. the potential for ICMP packets to receive different treatment in the network.

Another approach would be to capture the TCP test packets and then use the information from TCP timestamp option to determine delay; this would have required higher disk throughput than available and would use CPU cycles in a way that could alter the test.

Yet another approach would be to modify the kernel to periodically output the delay information it already has (for FAST TCP). This is desirable, and was, in fact, originally done in FAST TCP. However, conventional TCP senders do not output delay data and further, even for FAST TCP senders, since it is the FAST TCP implementation that is being tested, it is advantageous to have a method of obtaining delay data that does not involve FAST TCP code.

Since none of these approaches serves the needs of the test completely, and I could not find an already-existing network measurement tool that would measure delay in addition to throughput, a custom tool was developed for this test.

### 3.2 Custom Test Tool Design

The design criteria for the tool were as follows:

1. ability to conduct TCP throughput measurements,
2. ability to concurrently conduct delay measurements on the same packets,
3. portability and simplicity,
4. client/server architecture with a traditional UNIX dæmon running in the background,

5. machine-readable output suitable for plotting, and

6. ability to request given values for TCP window size and I/O block size from the client.

Criterion 3 suggests avoiding kernel modification; since Berkeley socket interface does not provide access to any timing information, it becomes necessary to embed such information in the data stream itself. The server listens to connections on a fixed TCP port number, and, in the traditional UNIX manner, forks a child to handle each connection, during which one testing session takes place. A testing session consists of an initial exchange of messages that configure the session and of the actual test data. The initial exchange of configuration information consists of the following steps:

1. the server prints a banner that identifies the protocol and protocol version and notifies the client whether the session is welcome,

2. provided that the session is welcome, the client sends a session proposal that specifies the requested TCP window size and I/O block size,

3. the server sends a session response that specifies that TCP window size and the I/O block size that would actually be used (a server tries to honor the request if possible, but might modify the values if, e.g., one of the requested values is too large for this server).

Once this initial exchange is over, the client sends zero or more blocks of test data. Upon receipt of each block, the server sends back the first 16 bytes of the block. In principle, these 16-byte "block headers" can be used arbitrarily by the client; in the current implementation, four bytes are used to embed a timestamp obtained prior to sending the block while the rest of the space is unused. The timestamp, echoed back by the server, allows the client to learn the round-trip time of the connection. It is believed that in most TCP implementations the additional 16 bytes of payload sent by the server in response to each block will not cause generation of an extra packet but will instead be tacked onto an outgoing TCP ACK packet; for the TCP implementations that were used in this experiment this conjecture was verified by packet capture. The delay measurement, thus, has a relatively low overhead: two calls to `gettimeofday` system call per block size. For fast networks and reasonable block sizes and in the absence of TCP timeouts, this round-trip latency measurement methodology therefore promises reasonably accurate results. The choice of a block size is governed by the following considerations: on the one hand, a smaller block size results in a larger overhead, both in `gettimeofday` system calls and in `write` system calls; on the other hand, a larger block size results in larger serialization delay, thus skewing the measurement results towards larger numbers. In no case should the block size be smaller than the TCP maximum segment size; lower overhead and better performance is promised by page-aligned blocks; the reasonable range of block sizes appears to be between 8192 and 65536 bytes. Maximum value for the TCP window size and for the I/O block size on the server and desired values for these quantities on the client are configurable with command-line options.

This small tool (less than 1000 lines of code in version 0.2), and its source code, is made publicly available [6] under a BSD-style license in the hope that others might find it useful for their network testing needs. The exact protocol used in the measurements is specified in a separate file in the distribution tarball.

## 4 Test Execution

The tests were performed on November 11, 2003 between 2 AM and 6 AM Eastern Time (the local timezone for the bottleneck link). The choice of time was dictated by the desire to shield daytime production users from any potential adverse effects of the experiment. It should be noted that even during the test window, in excess of 200 Mb/s of production traffic was transported by the bottleneck link. The command line used on the clients (senders) was "`i2perf -l65536 -w4194304 -i1 -t`⟨*duration*⟩ ⟨*server_ip*⟩"; this requests a TCP window size of 4 MB and I/O block size of 64 kB.

| Path | Mb/s | min RTT | avg RTT | max RTT |
|---|---|---|---|---|
| PATH1 | 286.100 | 58.751 | 61.419 | 1868.287 |
| PATH2 | 938.460 | 44.250 | 62.850 | 802.687 |
| PATH3 | 650.589 | 59.378 | 100.497 | 1600.684 |
| PATH4 | 602.295 | 46.129 | 91.357 | 851.856 |

Table 1: Overall baseline session results (RTT is in units of ms).

| Path | Mb/s | min RTT | avg RTT | max RTT |
|---|---|---|---|---|
| PATH1 | 280.411 | 58.643 | 61.143 | 1111.416 |
| PATH2 | 747.636 | 25.068 | 77.155 | 1090.875 |
| PATH3 | 574.248 | 58.788 | 111.707 | 3604.343 |
| PATH4 | 577.297 | 50.610 | 96.492 | 957.372 |

Table 2: Overall concurrent session results (RTT is in units of ms).

Initially, four 15-minute baseline tests were conducted, one for each path, with no competing test traffic (but against production traffic background). These tests verified that, on each Linux path TCP was capable of performing well and getting a significant fraction of 1 Gb/s and, on the FreeBSD path obtained throughput and delay against which to compare throughput and delay numbers obtained later while running with a congested path.

The baseline tests were followed by the experiment where four flows were run concurrently, with 15-minute staggered start (first the flow on PATH1 was started, 15 minutes later—the flow on PATH2, etc.). The flows on the Linux paths (PATH2, PATH3, and PATH4) ran for one hour, while the flow on the FreeBSD path (PATH1) ran for two hours to completely cover all the other tests.

## 5 Test Results

The overall session results for the baseline tests and the concurrent test are presented in tables 1 and 2, respectively. As can be seen, during the concurrent test, the throughput on PATH2 and PATH3 (the FAST TCP paths) has decreased significantly, while the decrease on PATH1 and PATH4 (the conventional TCP paths) has not been significant. In other words, when a bottleneck was congested by a mixture of FAST and Reno TCP flows, the bulk of the penalty associated with congestion went to the FAST flows; this is to be expected: as the output queue on the bottleneck network interface starts to build up[1], the FAST flows should sense the onset of congestion based on the increase in round-trip delay and scale back their congestion window size, while the Reno flows, seeing no losses and, hence, unaware of the fact that the bottleneck is running at full utilization, would continue sending without halving their congestion window (it should be noted that since the delay at the bottleneck increases by a few milliseconds, the Reno flows would obtain somewhat lower throughput for any given window size; this is exactly what we observe[2]).

The various plots of throughput and round-trip time obtained in user space are produced with the aid of the custom tool discussed above. The reporting interval was set to 1 second; the throughput is thus 1-second average, while the minimum and average RTT are taken from the 1-second sample. The plots of RTT for PATH2 and PATH3 obtained in kernel space are taken from the periodic reports about connection progress that FAST TCP outputs to SYSLOG; these are the internal kernel state parameters: for the version of FAST TCP used in this experiment, the base RTT is the minimum RTT observed since connection start, while the average RTT is an exponentially decaying average. Note that the minima and the averages obtained in user space and in kernel space are produced using different algorithms[3] and,

[1]The bottleneck interface in question was configured with drop-tail queuing discipline and the amount of buffer space on it was quite significant: four identical OC-48 interfaces share about 8 seconds worth of buffer space between them and, since there was no congestion on the other three interfaces during the experiment, the delay would need to increase to 8 seconds before router drops would occur. The delay never increased this much during this experiment.

[2]It is notable that, while the flow on PATH1 is limited by the CPU on its FreeBSD sender, it still experiences corresponding changes in throughput with changes in delay.

[3]The user space measurements use averaging over a longer period of time, and the average is not weighted. But more importantly, not only the samples of timestamps account for different time intervals, the measures are of different quanti-

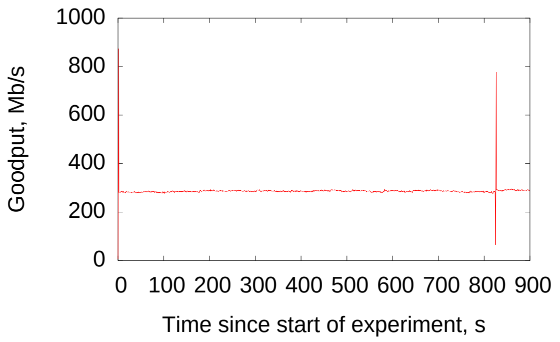

Figure 4: Baseline throughput of PATH1 (stock FreeBSD); average is 286 Mb/s.

therefore, are not directly comparable: one cannot be used to validate the other; rather, they complement each other to provide both the information about delays experienced by user traffic and about the kernel state.

Figures 4, 5, 6, and 7 present the changes of throughput during baseline tests on PATH1, PATH2, PATH3, and PATH4, respectively, as a function of time. On Fig. 4 (the FreeBSD path) the connection is CPU-limited at the sender; the first spike is likely to be a slow start artifact; the cause of the second spike around second 820 is unknown. On Fig. 5 and Fig. 6 we see the typical FAST TCP performance: a very stable throughput, close to the link capacity on PATH2; I do not know why PATH3 obtained only 651 Mb/s, but it should be noted that, statistically, this an exceptional level of performance when viewed among all flows that traverse Abilene [4]. On Fig. 7 (stock Linux), we observe the characteristic congestion-limited Reno TCP sawtooth.

Figures 8, 9, 10, and 11 represent the corresponding evolution of the delay on PATH1, PATH2, PATH3, and PATH4, respectively. Fig. 8 shows delay that's as stable as throughput on Fig. 4; thus, PATH1, where TCP is used essentially as a delay and loss measure-

ties: user-space measurements account for complete internal queues, while kernel-space measurements only account a fraction of the internal queues that depends on where the packet is timestamped within the kernel.

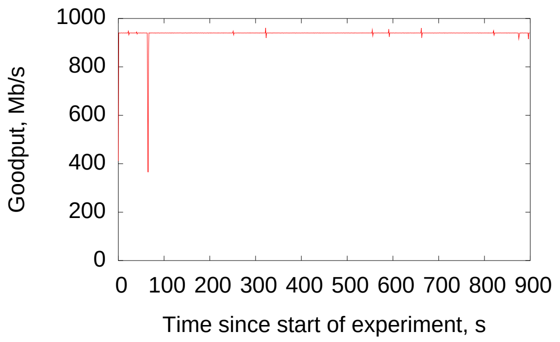

Figure 5: Baseline throughput of PATH2 (FAST); average is 938 Mb/s.

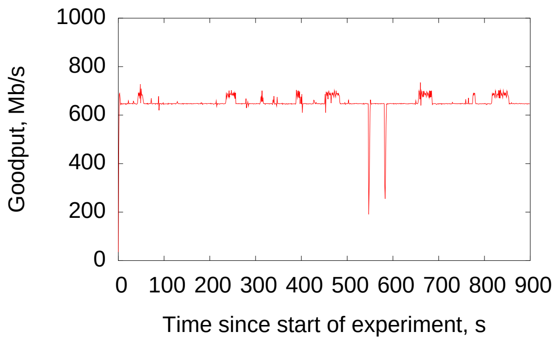

Figure 6: Baseline throughput of PATH3 (FAST); average is 651 Mb/s.

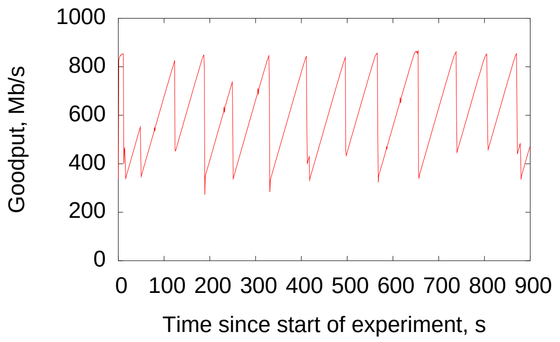

Figure 7: Baseline throughput of PATH4 (stock Linux); average is 602 Mb/s.

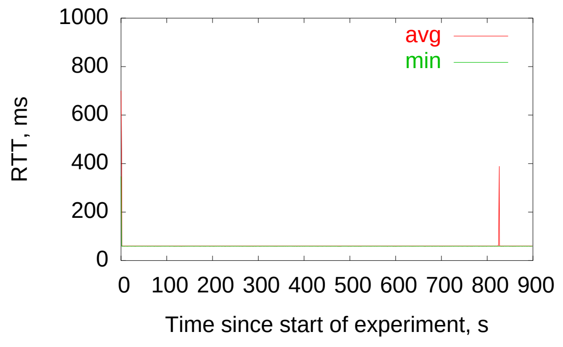

Figure 8: Baseline round-trip time of PATH1 (Reno), measured in user space; minimum is 59 ms; average is 61 ms.

ment tool, makes for an good way to evaluate what would happen to production traffic when the link is congested. On Fig. 9 and Fig. 10 we see one of the regimes in which FAST TCP can operate: the difference between the minimum and the average delay is fairly stable and significant; one can assume that the delay-based algorithm is trying to drive the delay to the given target, so the bottleneck queue (inside the host) is oscillating. Note also the kernel notion of delay on Fig. 12 and Fig. 13; unfortunately, the FAST kernel did not give any indication of dispersion of delay times (only the exponentially decaying average) so it is difficult to deduce from the available data whether the control algorithm is working as it should and converges to a stable value of the exponentially decaying average. Fig. 11 complements our understanding of the sawtooth by giving us an indication of the bottleneck queue length before a loss occurs; note that from the available data we still cannot conclude whether the loss is caused by overrunning the Linux transmit queue (controlled by `txqueulen` parameter of `ifconfig`) or by something else, such as the increased processing overhead of a longer queue or perhaps some aspect of the driver behavior; overrunning the queue appears to be the more likely cause of loss.

Figures 14, 15, 16, and 17, finally, present the main data of this paper: the results of a concurrent run of

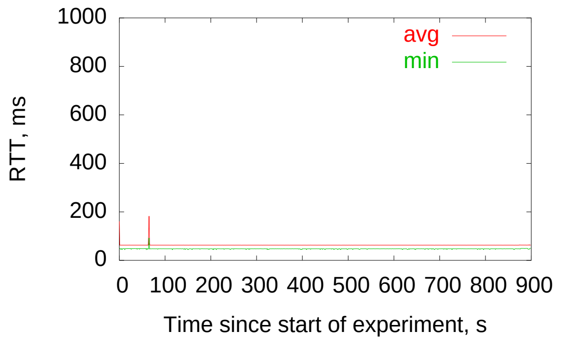

Figure 9: Baseline round-trip time of PATH2 (FAST), measured in user space; minimum is 44 ms; average is 63 ms.

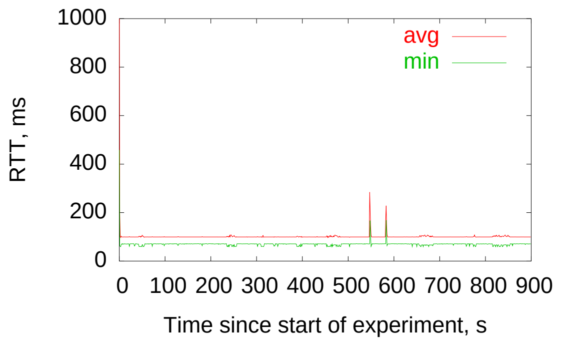

Figure 10: Baseline round-trip time of PATH3 (FAST), measured in user space; minimum is 59 ms; average is 100 ms.

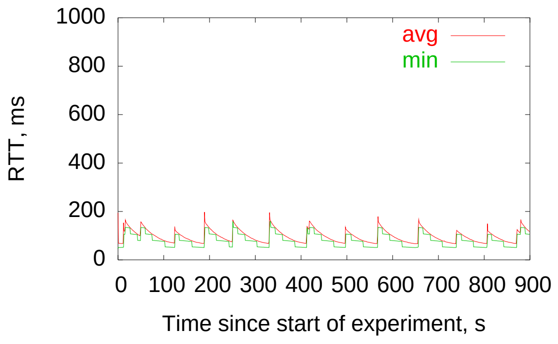

Figure 11: Baseline round-trip time of PATH4 (stock Linux), measured in user space; minimum is 46 ms; average is 91 ms.

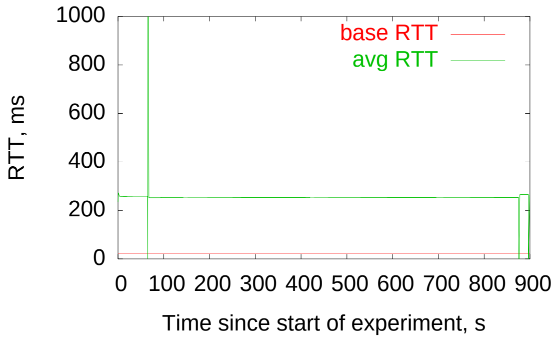

Figure 12: Baseline round-trip time of PATH2 (FAST), measured in kernel space.

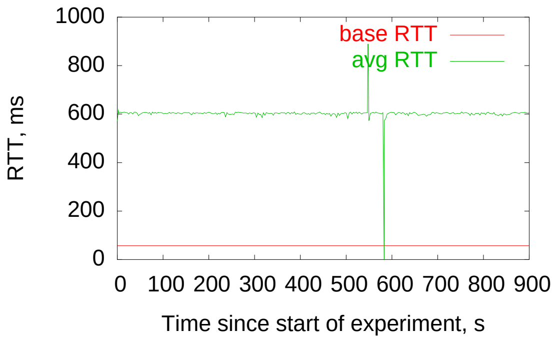

Figure 13: Baseline round-trip time of PATH3 (FAST), measured in kernel space.

all four flows that resulted in congesting the OC-48 bottleneck link with 2.5 Gb/s of traffic. On the 900-second baseline plots, each experiment started at its own time and none had overlapping running times; on all the 7200-second plots of concurrent runs, on the other hand, the same period is presented. The vertical hairlines, placed each 900 seconds, mark the beginning and ending times of the flows (table 3). The 30-minute interval from second 2700 to second 4500 is when all four streams are running and the network is congested. Fig. 14 depicts the throughput changes of the four flows. Fig. 15 and Fig. 16 show the round-trip time, measured in user space; on other RTT plots, the average and minimum RTT of the same flow are shown on the same plot, but in this case they are split between two figures to avoid placing eight vigirously fluctuating lines on the same plot. Fig. 17 reveals the kernel idea of delay on PATH2 and PATH3, both using FAST TCP (these data are not available for conventional TCP paths).

An independent validation of the fact that the bottleneck link was congested during the experiment comes from SNMP data obtained from the SoX interface of the Abilene core router in Atlanta [3], see Fig. 18. Note that the utilization shown on that graph slightly exceeds the link capacity (2.5 Gb/s); this appears to be either an averaging or plotting effect within RRDTOOL or an artifact of the way SNMP data are propagated within the router from the line

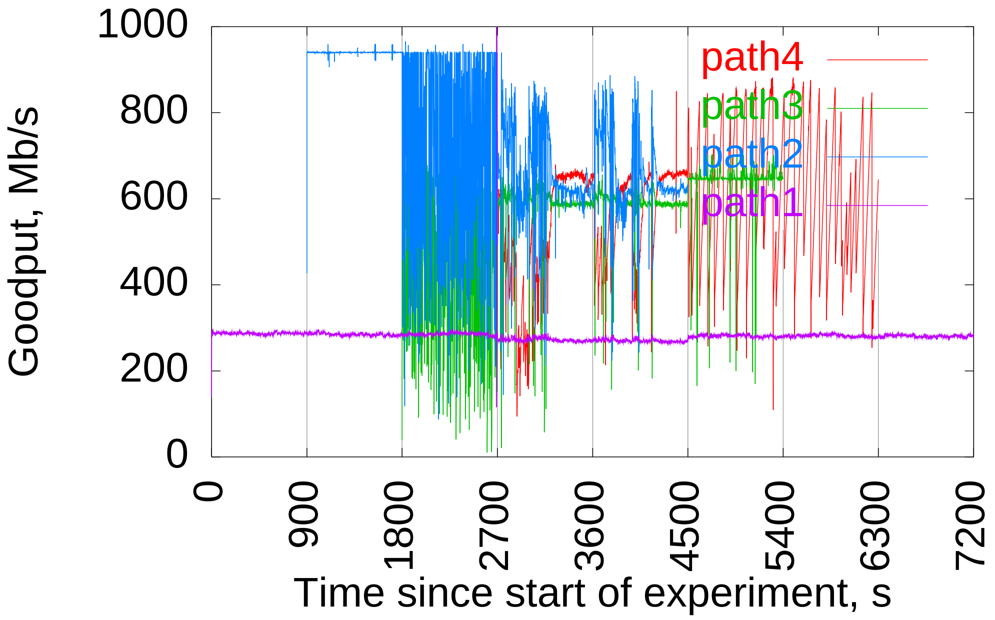

Figure 14: Throughput of all paths during the concurrent experiment.

| Second | Event |
|---|---|
| 0 | PATH1 starts |
| 900 | PATH2 starts |
| 1800 | PATH3 starts |
| 2700 | PATH4 starts |
| 3600 | |
| 4500 | PATH2 finishes |
| 5400 | PATH3 finishes |
| 6300 | PATH4 finishes |
| 7200 | PATH1 finishes |

Table 3: Timeline of the 2-hour concurrent test.

card to the routing engine.

## 6 Observations

1. SNMP data shows that the bottleneck link is congested;
2. Since PATH1 lines on Figures 14, 15, and 16 are essentially flat, the bottleneck link never loses even a single packet belonging to the PATH1 flow[4] and the PATH1 flow packets are not significantly delayed (raw data examination shows an increase of delay of about 5-6 ms);
3. While the baseline tests with FAST paths show moderate increases in observed delay with respect to the base delay, at time of congestion this delay becomes much greater, sometimes even exceeding 1 second; yet since the bottleneck link delay is insignificant, this delay must be coming from queuing sources within the FAST TCP hosts;
4. Similarly great is the measured delay (during the concurrent run) on PATH4 using conventional Linux TCP;

[4]Since PATH1 uses conventional Reno TCP, a loss would result in halving the congestion window—and, thus, the rate—and slowly ramping it back up. This never happens.

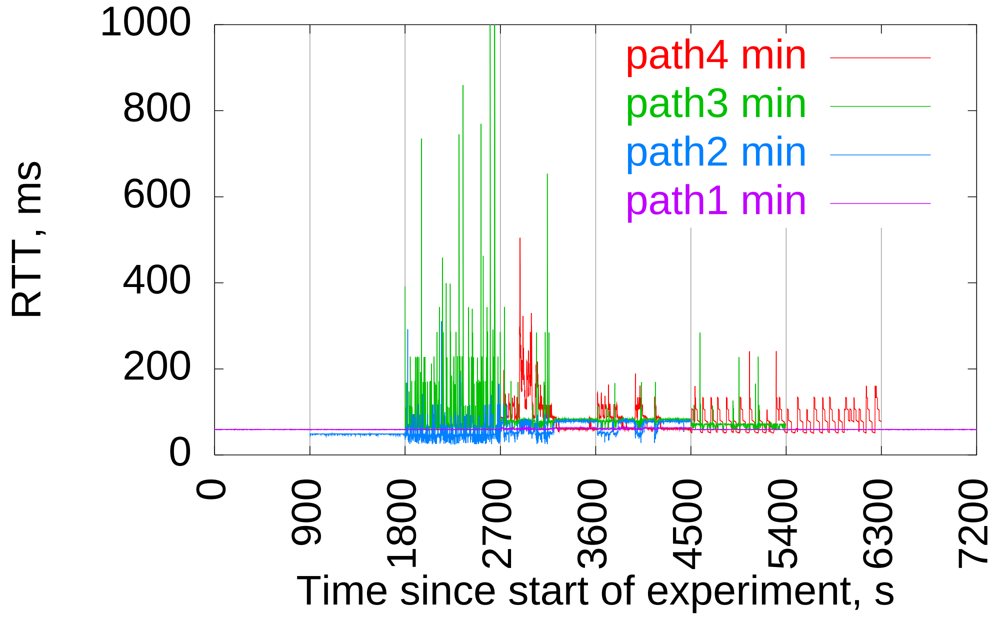

Figure 15: Minimum round-trip time of all paths during the concurrent experiment, measured in user space.

5. In the concurrent run, at time of congestion, the variability of throughput of FAST TCP is significantly less than at the time when FAST TCP flows share a non-congested bottleneck with other FAST or Reno flows.

## 7 Conclusions

1. FAST TCP in its present form appears to allow to perform high-speed data transfers that saturate the network without adversely affecting production traffic, either in terms of loss or in terms of delay—using simple FIFO queuing and without any need for quality-of-service techniques such as scavenger service [10] or any active queue management techniques.

2. The Linux TCP implementation[5] is prone to building and sustaining queues inside the hosts; the presence and extent of such undesirable queuing appear to depend subtly on such factors as existence of cross-traffic and transmit queue length. FAST TCP, being a set of kernel patches to Linux, appears to a certain extent to inherit this property. Some degree of internal queuing is normal due to buffering at various levels (`write` block size, IP queue, driver queue, NIC buffers); however, having hundreds of milliseconds worth of extra queuing delay serves no useful purpose.

## 8 Acknowledgments

The tests used FAST TCP machines owned and operated by Caltech, with the help of Steven Low, Cheng

[5]In other tests not described in this paper, internal queuing was also observed on FreeBSD. However, the phenomenon does not appear to be reproduceable on FreeBSD and usually only is manifested at the start of a connection, with the queue draining afterwards. This seems to suggest that on FreeBSD such queuing is related to slow start overshoot: FreeBSD remembers slow start threshold value in clone route entries, enabling subsequent runs to avoid the overshoot.

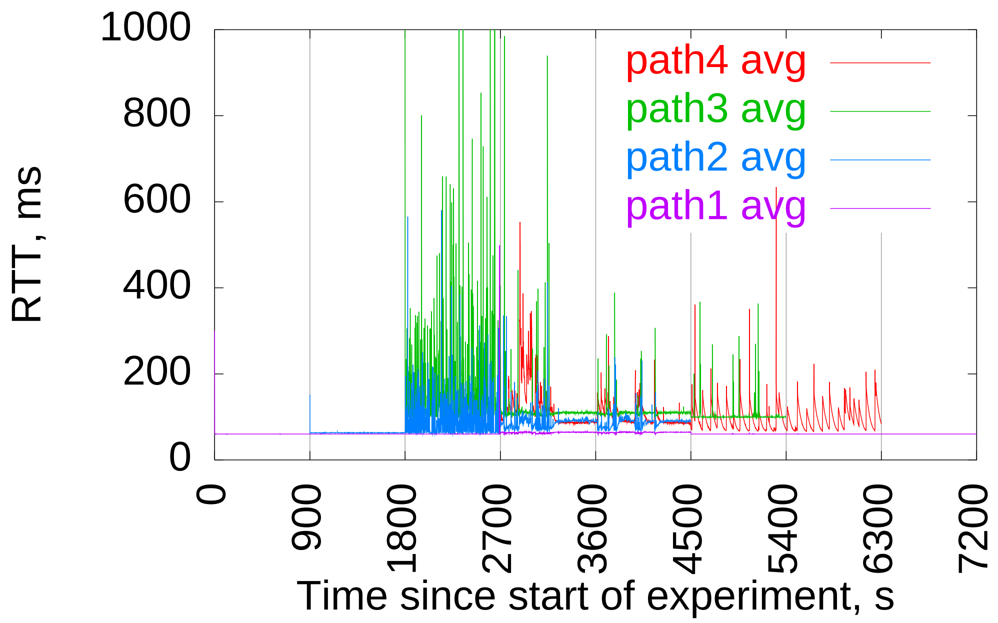

Figure 16: Average round-trip time of all paths during the concurrent experiment, measured in user space.

Jin, and Raj Jayaraman. Cas D'Angelo (Southern Crossroads GigaPoP), David Richardson (Pacific Northwest GigaPoP), Matt Mathis (Pittsburgh Supercomputing Center), and John Moore (North Carolina Internet2 Technology Evaluation Center) provided high-speed hosting for the test machines.

## References

[1] "Abilene Backbone Network," http://abilene.internet2.edu/

[2] "FAST Protocols for Unltrascale Networks," http://netlab.caltech.edu/FAST/

[3] "Indiana University Abilene NOC Weathermap," http://loadrunner.uits.iu.edu/weathermaps/abilene/

[4] "Internet2 NetFlow Weekly Reports," http://netflow.internet2.edu/weekly/

[5] C. Jin, D. X. Wei and S. H. Low, "FAST TCP: motivation, architecture, algorithms, performance," IEEE Infocom, March 2004

[6] S. Shalunov, "i2perf, network capacity tester," http://www.internet2.edu/~shalunov/i2perf/

[7] S. Shalunov, "FAST TCP tests over Abilene conducted at 2003-09-23T06:00Z," test log, http://www.internet2.edu/~shalunov/fast-tcp-abilene/

[8] S. Shalunov, "Testing FAST TCP over Abilene," presentation at FAST TCP project review, October 2003, Caltech, http://www.internet2.edu/~shalunov/talks/20031029-Caltech-FAST-Review.pdf

[9] "SoX: About Us", section "SURA Members," http://www.sox.net/about/main.htm

[10] "QBone Scavenger Service (QBSS)," http://qbone.internet2.edu/qbss/

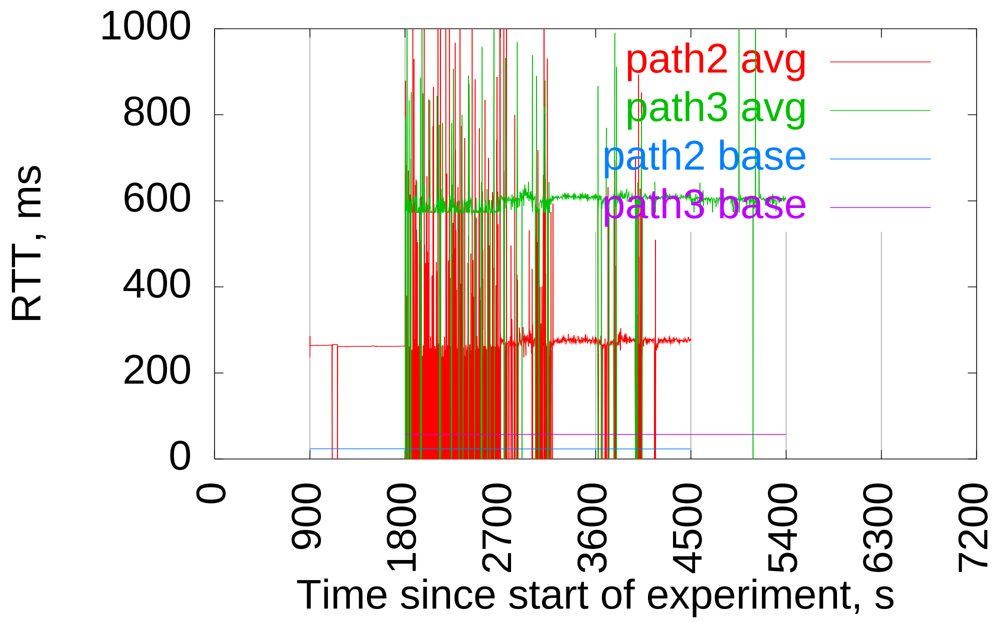

Figure 17: Round-trip time of PATH2 and PATH3 during the concurrent experiment, measured in kernel space.

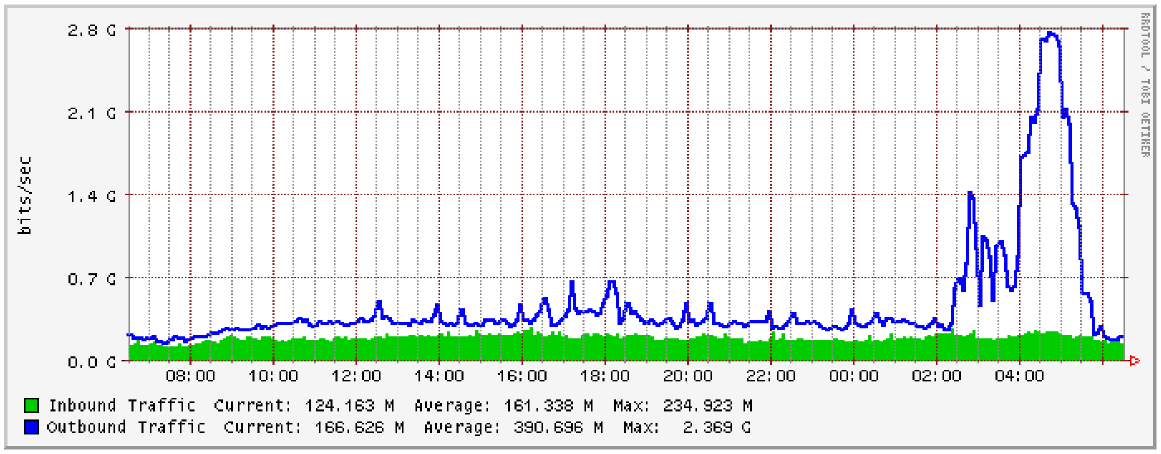

Figure 18: The RRDTOOL plot of link utilization (5-minute averages) on the SoX interface of the Abilene core router in Atlanta, viewed after the experiment was over.