A Comparison of GSIFTP and RFIO on a WAN

Rajesh Kalmady[1]
Brian Tierney[2]

Introduction

We present a comparison of wide-area file transfer performance of the new Globus GSIFTP and CERN's rfio tools.

Description of RFIO

RFIO (Remote File I/O) is one of the components that make up the CERN Advanced Storage Manager (CASTOR) [1]. RFIO implements a remote version of most standard POSIX calls like open, read, write, lseek and close using a very light weight protocol. The control and data streams are separated. To optimize the data throughput by overlapping network and disk I/O, a circular buffer and two threads are used for each connection. Multiple parallel streams are not implemented yet but could be done for use on a high speed WAN.

Client programs can use RFIO libraries to access files on remote disks or in the CASTOR namespace. The libraries detect the location of the files and take appropriate action to make them available for the client application.

Several RFIO equivalents of UNIX commands like cp, rm, mkdir etc. are also present.

Description of GridFTP

GSIFTP is a high-performance, secure FTP protocol, which uses the GSI (Grid Security Infrastructure), for authentication. It is a subset of the GRIDFTP protocol. GSIFTP is currently being enhanced with a variety of protocol features appropriate for grid applications, and integrated with the GASS client library. This will allow grid applications to have ubiquitous, high-performance access to data in a way that is compatible with the most popular file transfer protocol in use today, FTP. GSIFTP is being used as the basis for higher-level work on the Data Grid, and for managing and assuring application access to data on the grid.

The Globus [2] development tree contains the sources for the gsiftp libraries that are used by example client programs and the grid enabled Washington University ftp server. GSIFTP [3] has useful features such as TCP buffer sizing and multiple parallel streams.

[1] IT Division CERN, Computer Division, Bhabha Atomic Research Centre

[2] IT Division CERN, Computing Sciences Division, Lawrence Berkeley National Laboratory

TCP tuning and parallel sockets

TCP is used by various applications for reliable data transfer over a network. TCP was originally designed for providing reliable data delivery over an underlying unreliable network. From the time it was originally designed some 30 years back, TCP has been performing remarkably well over a wide selection of networking technologies from LANs to WANs leading to the Internet revolution. But it has been found that TCP is not able to adapt well to the new high speed networking technologies and recently there has been research addressing this issue. TCP performance depends not only on the bandwidth itself but also upon the product of the transmission rate and the round trip delay [4]. This bandwidth-delay product determines the amount of data that could fill the network link. To get the maximum throughput on a high speed WAN, TCP should be able to send so much of data down the pipe without waiting for an acknowledgement. This implies that so much of buffer space should be available to TCP at both ends of the pipe.

Most operating systems have ridiculously low defaults for the TCP send and receive buffer sizes. This may be suitable for operating on a standard local area network of 100Mbps, but not when the system is connected on a "long fat pipe" with large delays and high bandwidths. Therefore it is critical to tune the buffer size parameters to obtain optimal throughput.

TCP socket buffer size tuning is a convenient way of obtaining maximum throughput out of a high speed WAN link. But this has an obvious drawback of knowing the "correct" value for the TCP window size. The optimal window size needs to be calculated by accurate measurements of the delay and bandwidth of the link. Then the parameters of the TCP stack in the kernel have to be changed by the system administrators at both ends. After this a lot of fine-tuning needs to be done over a period of time from a few hours to possibly several weeks. This may not be possible with some operating systems, which do not provide hooks to modify these parameters. This has led to research on an alternative technique to obtain high throughput, viz. Parallel data streams [5].

From the names 'parallel streams' it is fairly obvious what it is all about. Instead of having just one TCP connection between the sender and the receiver, multiple connections are opened with multiple sockets in charge. Data to be sent are split into 'n' partitions and fed down the 'n' connections through the 'n' sockets. At the receiver end, these data partitions are reassembled back. Since the TCP send and receive buffers are specific to one socket, having parallel sockets effectively increases the buffer sizes without any change to the kernel parameters.

Test Environment

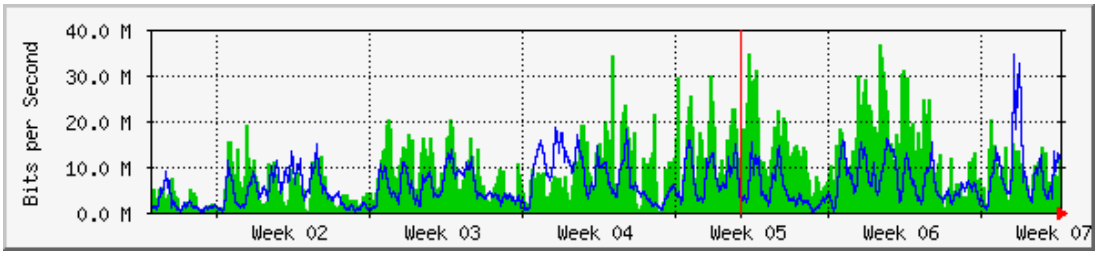
The test environment consisted of a 45 Mbps link between CERN and ANL with a RTT of 125 ms. We started out with CERN-LBL link but switched over to CERN-ANL because it had less fluctuation.

GSI enabled WU-ftpd server version 0.4b6 was used as the test server. Test programs ‘extended_get’ and ‘extended_put’ from the globus distribution were the clients chosen. These programs test the parallel stream and buffer tuning features of GSIFTP.

Test Methodology

We used the programs extended_get and extended_put from the globus distribution as the test clients. 4 files of sizes 1 MB, 25 MB, 50 MB and 100 MB were transferred over the network. The number of times the files were transferred depends on the file size. We took 8-10 readings for the 1 MB file down to 2 or 3 readings for the 100 MB file. This is because small data transfers are more susceptible to transient network glitches. After the readings were taken, we threw out the too high and too low values and took the average of the remaining. This gave a fairly accurate value for the transfer rate. Measurements of setup time (which includes security and other protocol overhead) and transfer time were taken.

This method worked fine for the untuned buffers and the task was generally over in a couple of days. But there were a lot of problems when the readings for the tuned buffers were taken. Data transfers with tuned buffers are very susceptible to slight changes in the network traffic. This resulted in whole new sets of values being obtained on different days. We had to wait for those precise days when the network behaviour was consistent with the partial readings already taken. Shown below is a snapshot of the network traffic through the CERN internet links for the last six weeks.



Results

Setup time:

The following table shows average setup time for RFIO Get/Put and GSIFTP Get/Put

	RFIO GET	RFIO PUT	GSIFTP GET	GSIFTP PUT
Default Buffers	0.659857	0.661024	3.00091	2.94608
Tuned Buffers	0.644506	0.87232	2.98879	3.27761

Setup time is the time from the start of the client application to start of the data transfer. This includes any application setup, connection and authentication. Obviously RFIO has much less setup time than GSIFTP, as the protocol is much simpler.

Get Results

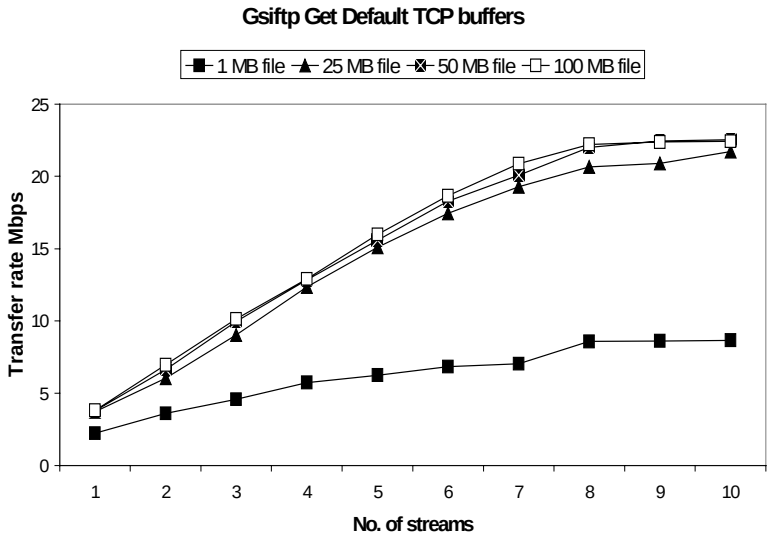


Figure 1

Figure 1 illustrates the variation in transfer rate achieved with increasing number of parallel streams with GSIFTP. These figures are with the default TCP buffers that are typically 64 KB in the test environment. Four different files were transferred with sizes of 1MB, 25MB, 50MB and 100MB. The graph shows the curves for the larger files going up pretty linearly with the number of streams, reaching a peak at around 23 Mbps for 9 streams.

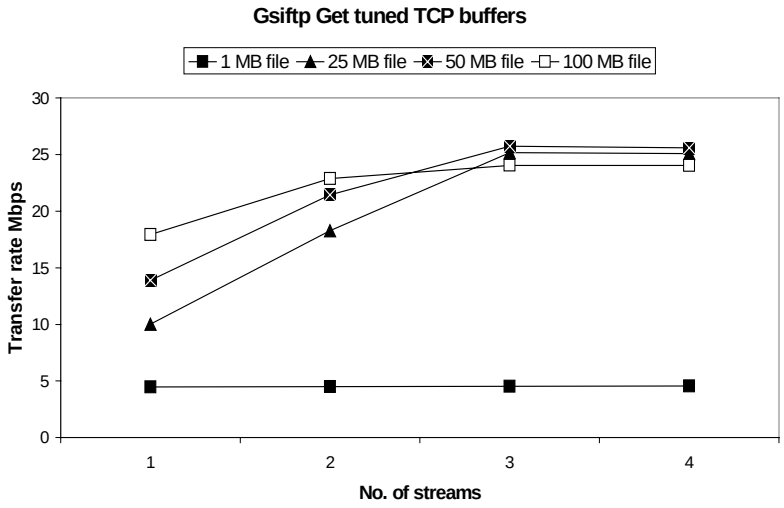


Figure 2

Figure 2 shows the same with TCP buffers tuned to 1 MB. This curve is pretty similar to the earlier one except that the peak is attained with just 3 streams. During the course of the testing we have found that the performance of gsiftp with tuned

buffers varies drastically with variations in the network traffic. The above peak has occurred at single, 2 , 3 or 4 streams on different days. Sometimes we have seen that the performance drops after reaching the peak causing the above curve to look like a inverted ‘V’ or a sine wave. One such plot with the put operation is shown later in this report. In short, the performance is very unpredictable.

Put Results

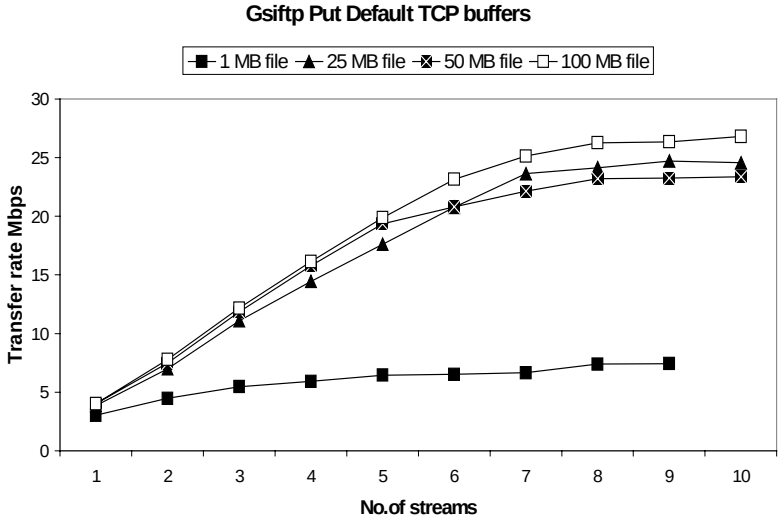


Figure 3

Figure 3 illustrates the results for GSIFTP Put with default buffers. The graph looks very similar to its Get counterpart with the obtained transfer rate increasing linearly until reaching a plateau.

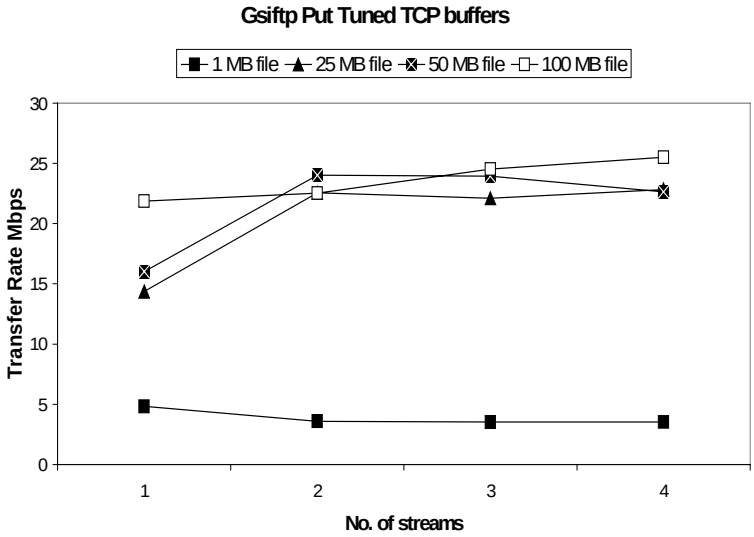


Figure 4

Figure 4 shows the GSIFTP Put results with tuned buffers. Again it looks similar to the Get plot with the plateau occurring at around 25 Mbps.

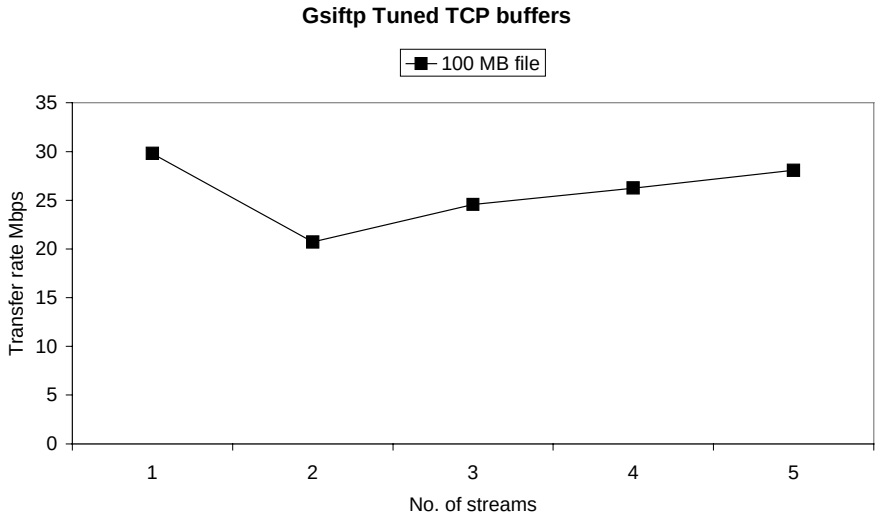


Figure 5

The above figure is an interesting plot of the performance of Gsiftp put with tuned buffers for a 100 MB file. This and the previous plot were taken on the same machines with same software, but on different days! Whereas plot 4 shows the performance increasing with the number of streams until it reaches a plateau, plot 5 shows peak performance at single stream, a sharp drop with 2 and a slow increase with more streams. This is just one of the many scenarios obtained with the tuned buffers. This clearly illustrates the tricky issues involved in tuning TCP buffers.

Comparison of RFIO Get and GSIFTP Get

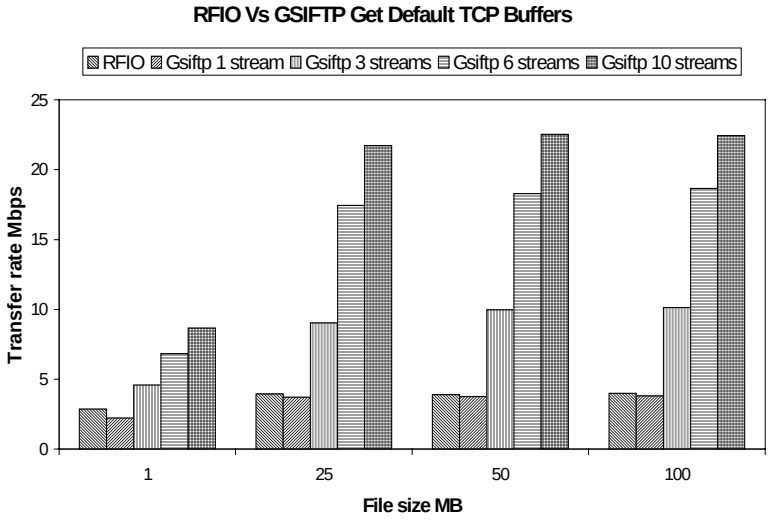


Figure 6

The above bar chart shows the comparison of RFIO with GSIFTP with multiple parallel streams. We find that the performance of RFIO is better than GSIFTP for 1 stream, but after that GSIFTP gets better and better with more parallel streams.

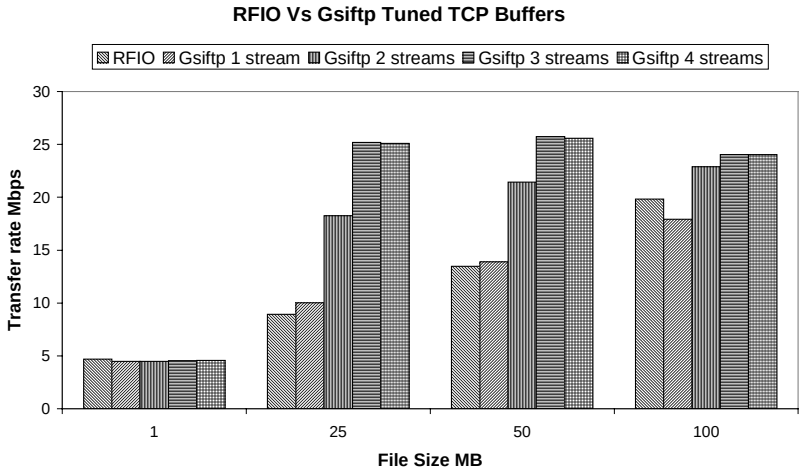


Figure 7

This bar chart shows RFIO and GSIFTP performance with tuned buffers. With tuning RFIO comes pretty near GSIFTP, which shows that proper tuning makes a lot of difference.

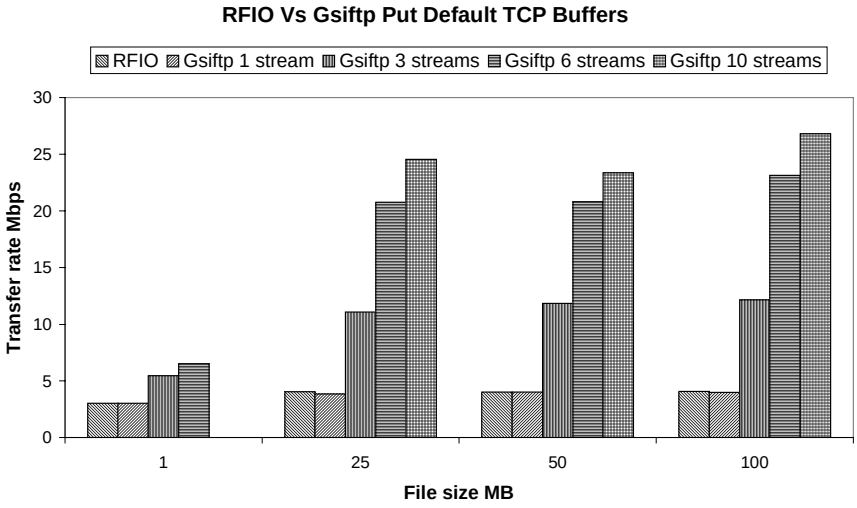


Figure 8

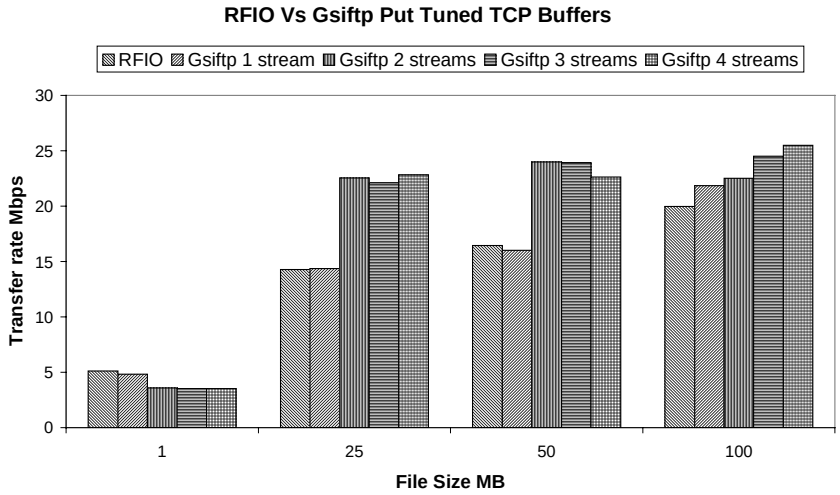


Figure 9

The above two barcharts are the corresponding comparison charts between RFIO and Gsiftp for the put operation.

Comparison with iperf

'iperf' is a tool for measuring the performance of a network under various conditions. It is possible to set different TCP buffer sizes and use multiple parallel streams. It is a straightforward transfer of data from the client to the server with minimum overhead. Gsiftp compares well with iperf. On the CERN-ANL link the best bandwidth obtained with 'iperf' was about 32 Mbps with 20 parallel streams and 64KB buffer size. For the same parameters Gsiftp performed at 29 Mbps. So the overheads associated with Gsiftp are quite minimal.

Use of Netlogger

We have used the Netlogger[6] toolkit extensively during the initial work with RFIO and GSIFTP. Netlogger calls were inserted in interesting places in the code to generate timestamps of various events. These logs were then examined using NLV. We uncovered two major bugs in GSIFTP, which were then corrected by the Globus team.

Sample Netlogger plots for RFIO and GSIFTP are shown below with descriptions.

Figure 10 is the Netlogger plot for an RFIO get operation. There are three sections in this plot. The topmost section is the Netlogger output from the server. The top line is from the produce thread in the server that reads a chunk from the file. The lower line is the sender thread in the server which send the chunk of bytes over the socket. The middle section is the corresponding Netlogger output from the rfcp client. The bottom section consists of the scatter plot indicating the sizes of the various packets received by the client over the network. We have observed that most of the

packets received are in the range of 1400 bytes and occasionally a packet of 16 KB to 128 KB comes across.

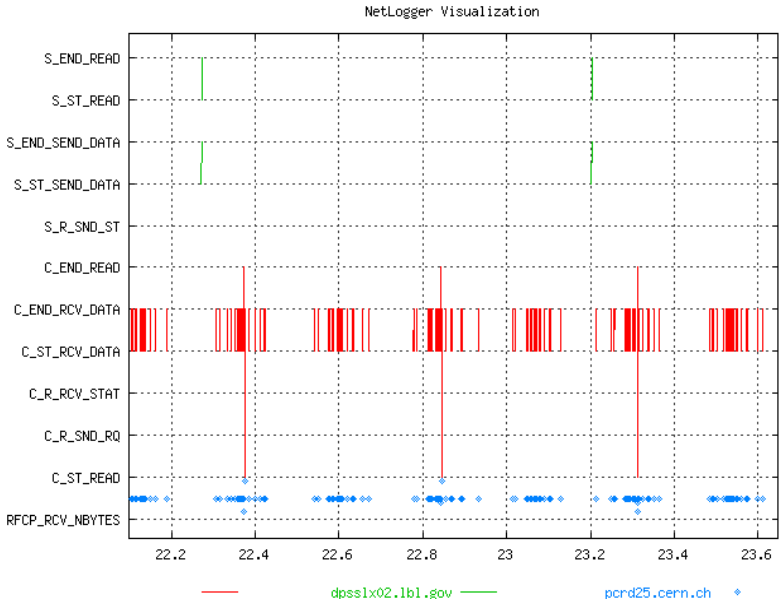


Figure 10

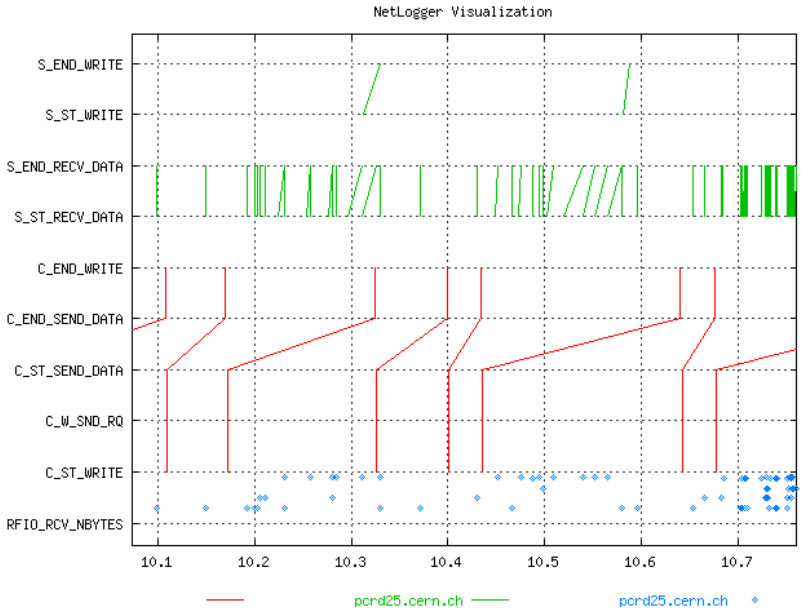


Figure 11

Figure 11 shows the Netlogger plot for an RFIO put operation. Here we see the client sending data to the server in large chunks and the server receiving it small packets. Intermittently the consume thread of the server gets scheduled and it writes a block of data to the file.

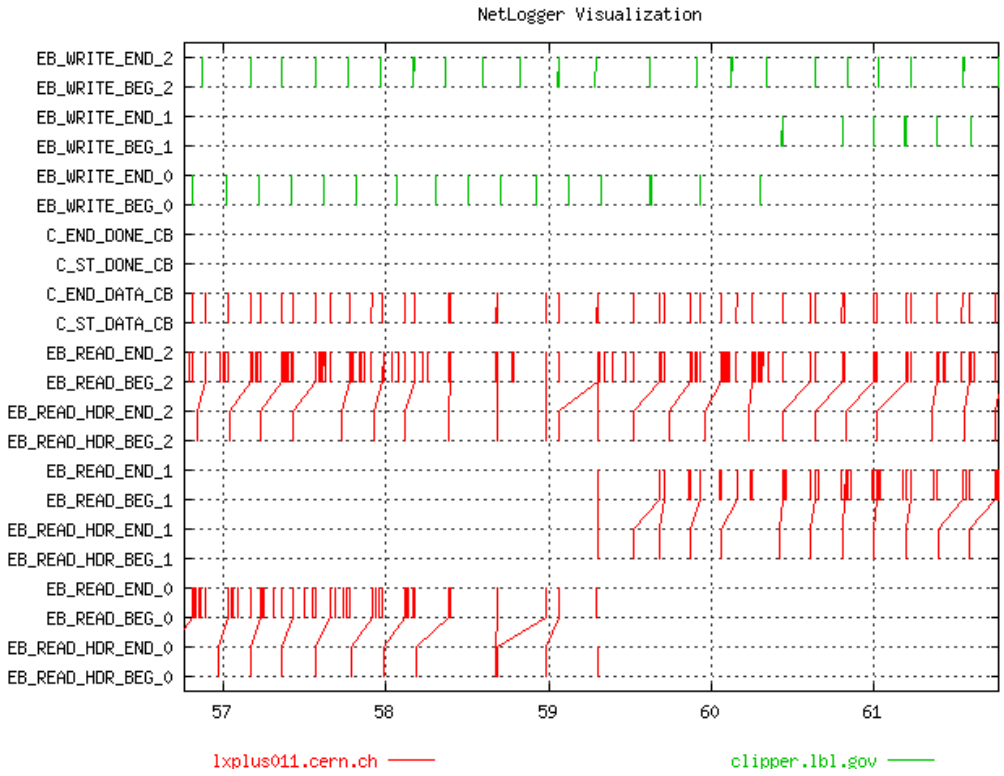


Figure 12

Figure 12 shows the netlogger plot of a broken version of the gsiftp software. We observed that even when 3 parallel streams were requested, data used to be transferred on any 2 of the 3 streams with streams being switched in between. In the above plot, data arrives on streams 0 and 2 initially and switches to 0 and 1 midway. This was reported to the globus team and the bug was fixed. Figure 13 shows the plot after the bugfix.

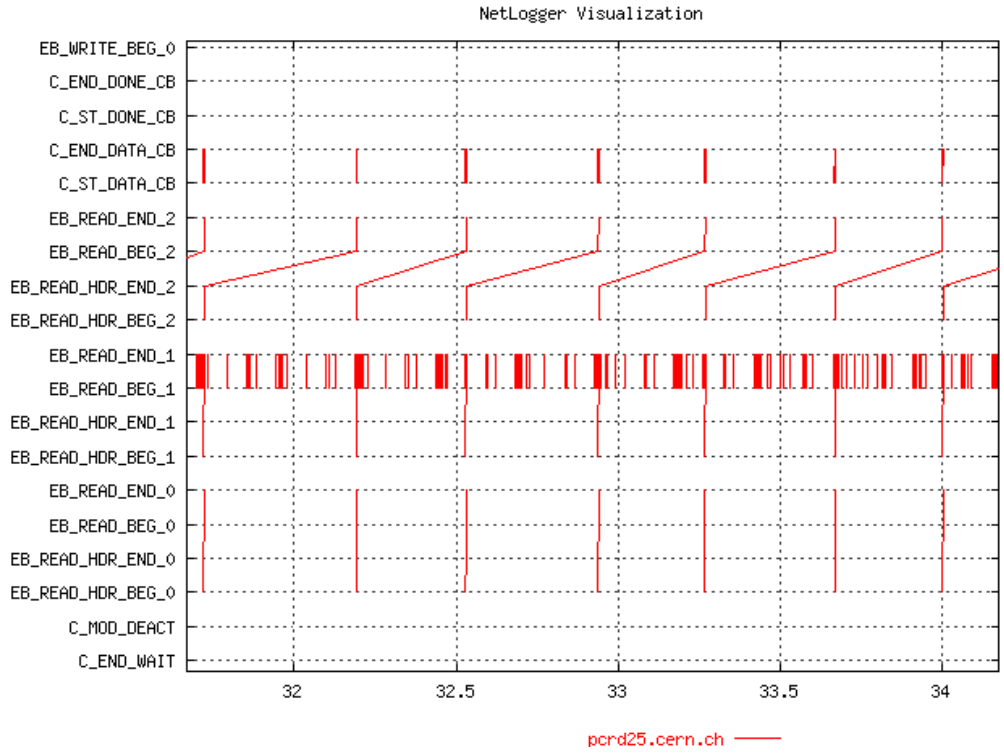


Figure 13

Interesting Linux OS issues

There were 2 interesting Linux issues that came to light during the course of our testing.

1. Linux 2.2 RTO bug

We observed that miserable throughput was obtained when data was sent by TCP from a Linux sender to a Solaris receiver on a WAN link. Linux-Linux and Solaris-Linux transfers did not suffer from this problem. The problem was traced to a wrong computation of RTO (retransmit timeout) by the Linux sender. Linux apparently sets its RTO to a very low value (according to the standard) and expects to receive acks within this time period [7]. Solaris uses a larger delayed ack (about 50 ms) thus triggering a lot of retransmits over the line. This condition gets aggravated when data flies over a long fat network when retransmits are very costly. On the CERN-LBL link we found that Linux-Solaris transfer rate was 0.5 Mbps whereas the other permutations gave a rate of 25 Mbps. On the ANL-LBL line the condition was a bit better (3 Mbps and 50 Mbps). This was reported to the Linux developers and the bug is fixed in kernels 2.2.18 upward and 2.4.0-test12 upward.

2. Linux 2.4 autotuning

This is a new feature that has been introduced in Linux kernel 2.4. The TCP buffer sizes are initially set to a low value, typically 16 KB. As the data transfer takes place, the buffer size is continuously readjusted so as to obtain the maximum throughput. We have observed that the buffer size rises from the initial 16 KB to

around 256KB on the CERN – Berkeley line. Sometimes we have seen the buffer sizes go upto 1.4 MB, though this is a rare occurrence. The optimum buffer size for the CERN – LBL link is about 1.4 MB with which we get throughputs of about 25-30 Mbps. The autotuning algorithm usually sets the buffer size to about 200-400 KB, which delivers a throughput of about 6 Mbps only. Of course, this algorithm must be in its nascent stage and doubtless it will get better and better in the future and compute the buffer size correctly.

Conclusions

From these tests we learned the following:

1. Proper TCP buffer size setting is the single most important factor in achieving good performance. The performance obtained from 10 streams with untuned buffers can be achieved with just 2-3 streams if the tuning is proper.
2. 2-3 parallel streams will gain an additional 25% performance over a single tuned stream.
3. Data transfer with tuned buffers is highly sensitive to the variation in ambient network traffic. There should be a mechanism of dynamically varying the buffer sizes during data transfer.
4. It is possible to get the same throughput as tuned buffers using untuned TCP buffers with enough parallel streams.
5. Netlogger is a very valuable tool for performance analysis and application debugging.
6. Performing these types of tests over a busy network is a difficult and time consuming task. This problem gets aggravated with tuned buffers which is highly sensitive to variations in network traffic.

References

1. CASTOR (CERN Advanced Storage Manager) Project http://wwwinfo.cern.ch/pdp/castor/
2. The Globus Project http://www.globus.org/
3. GSIFTP http://www.globus.org/gsiftp-alpha/
4. TCP Tuning Guide for Distributed Application on Wide Area Networks, http://www-didc.lbl.gov/tcp-wan.html
5. PSockets: The Case for Application-level Network Striping for Data Intensive Applications using High Speed Wide Area Networks, http://www.ncdm.uic.edu/html/body_sc2000.html
6. NetLogger: A Methodology for Monitoring and Analysis of Distributed Systems, http://www-didc.lbl.gov/NetLogger/
7. Information on critical Linux TCP bug for high-speed WAN applications, http://www-didc.lbl.gov/Linux-tcp-bug.html