

A Comparison of GSIFTP and RFIO on a WAN

Rajesh Kalmady¹
Brian Tierney²

Introduction

We present a comparison of wide-area file transfer performance of the new Globus GSIFTP and CERN's rfio tools. The full version of the paper is available on the web at the following URL: <http://grid-data-management.web.cern.ch/grid-data-management/docs/GridFTP-rfio-report.pdf>

Description of RFIO and GSIFTP

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. Multiple parallel streams are not implemented yet but could be done for use on a high speed WAN.

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.

TCP tuning and parallel sockets

TCP performance depends not only on the available bandwidth, but upon the TCP window buffer size set in the Operating System [4]. The 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. 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. 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].

¹ IT Division CERN, Computer Division, Bhabha Atomic Research Centre, rajesh@magnum.barc.ernet.in

² IT Division CERN, Computing Sciences Division, Lawrence Berkeley National Laboratory, bltierney@lbl.gov

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 and Methodology

The test environment consisted of a 45 Mbps link between CERN and ANL with a RTT of 125 ms. 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.

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.

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 security protocol is much simpler.

Get results

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 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

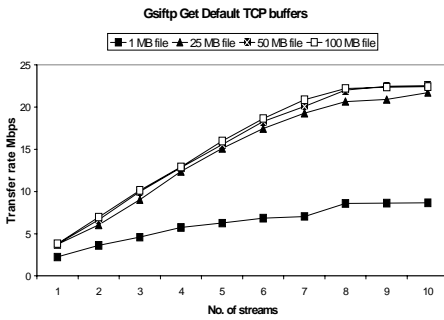


Figure 1

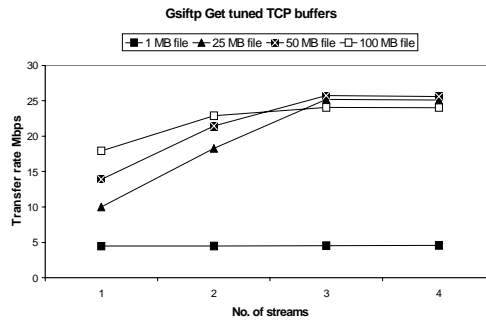


Figure 2

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 an inverted 'V' or a sine wave. In short, the performance is very unpredictable. The results for the put operation are similar to the get.

Comparison of RFIO Get and GSIFTP Get

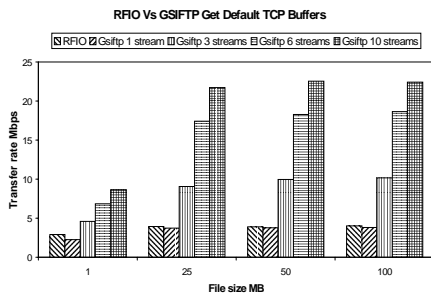


Figure 3

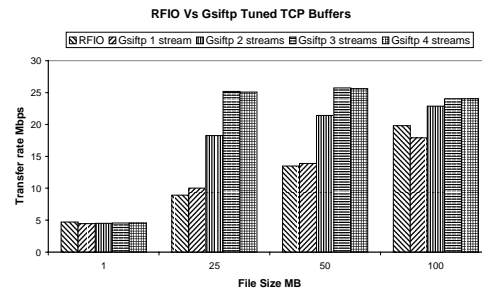


Figure 4

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.

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.

The put operation yields pretty similar results.

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, the Netlogger Visualization tool. We uncovered two major bugs in GSIFTP, which were then corrected by the Globus team.

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>