GridFTP Pipelining

John Bresnahan,[1,2,3] Michael Link,[1,2] Rajkumar Kettimuthu,[1,2] Dan Fraser,[1,2] Ian Foster[1,2,3]

[1]Mathematics and Computer Science Division
Argonne National Laboratory, Argonne, IL 60439
[2]Computation Institute
The University of Chicago, Chicago, IL 60637
[3]Department of Computer Science
The University of Chicago, Chicago, IL 60637

{bresnaha, mlink, kettimut, fraser, foster}@mcs.anl.gov

Abstract—GridFTP is an exceptionally fast transfer protocol for large volumes of data. Implementations of it are widely deployed and used on well-connected Grid environments such as those of the TeraGrid because of its ability to scale to network speeds. However, when the data is partitioned into many small files instead of few large files, it suffers from lower transfer rates. The latency between the serialized transfer requests of each file directly detracts from the amount of time data pathways are active, thus lowering achieved throughput. Further, when a data pathway is inactive, the TCP window closes, and TCP must go through the slow-start algorithm. The performance penalty can be severe. This situation is known as the "lots of small files" problem. In this paper we introduce a solution to this problem. This solution, called pipelining, allows many transfer requests to be sent to the server before any one completes. Thus, pipelining hides the latency of each transfer request by sending the requests while a data transfer is in progress. We present an implementation and performance study of the pipelining solution.

Index Terms— data transfer, small files, GridFTP

1 Introduction

GridFTP [1] is a well-known and robust protocol for fast data transfer on the Grid. Given resources, the GridFTP implementation provided by the Globus Toolkit [2] can scale to network speeds and has been shown to deliver 27 Gb/s on 30 Gb/s links [3]. The protocol is optimized to transfer large volumes of data commonly found in Grid applications [4][5][6][7]. Datasets of sizes from hundreds of megabytes to terabytes and beyond can be transferred at close to network speeds by using GridFTP.

Given the high-speed networks commonly found in modern Grid environments, datasets less than 100 MB are too small for the underlying protocols like TCP to utilize the maximum capacity of the network. Therefore, GridFTP – and most bulk data transfer protocols – experiences the highest levels of throughput when transferring large volumes of data.

Unfortunately, conventional implementations of GridFTP have a limitation as to how the data must be partitioned to reach these high-throughput levels. Not only must the amount of data to transfer be large enough to allow TCP to reach full throttle, but the data must also be in large files, ideally in one single file. If the dataset is large but partitioned into many small files (on gigabit networks we consider any file smaller than 100 MB as a small file), the performance of GridFTP servers suffers drastically

This problem is known as the "lots of small files" (LOSF) problem. In this paper we study the LOSF problem and present a solution known as *pipelining*. We have implemented pipelining in the Globus Toolkit, and we present here a performance evaluation of that implementation.

The rest of this paper is as follows. After discussing related work in Section 2, we provide details in Section 3 about the LOSF problem. In Section 4, we describe our pipelining solution, and in Section 5 we discuss the implementation of the proposed solution. In Section 6, we present experimental results. We conclude in Section 7 with a brief discussion of future work.

2 Related Work

To work around the LOSF problem, users often tar [8] into a single file all of the files they plan to send and then transfer that single file. This process requires additional CPU time and disk space. A similar but better approach is to tar all of the small files on the fly. This approach, which is taken by [9], does require additional network overhead and processing time, but very little. The main problem with this approach is the awkwardness of specifying which files will be part of the transfer set. If not all of the files are in the same subdirectory, the expression can be complicated.

Mode X is part of the GridFTP version 2 protocol [10]. It adds a notion of transfer IDs to data channel messages. This allows for many file transfers to occur at the same time. With this approach we could perform many transfers concurrently, giving the appearance of a single large file transfer. Although this is significantly more complicated than the solution we present here, the approach has potential. Unfortunately, to our knowledge, there is no existing or widespread implementation of GridFTP version 2, and we are looking for

an immediately usable solution.

3 LOSF Problem

The GridFTP protocol is a backward-compatible extension of the legacy RFC959 FTP protocol [11]. It maintains the same command/response semantics introduced by RFC959. It also maintains the two-channel protocol semantics. One channel is for control messaging (the control channel) such as requesting what files to transfer, and the other is for streaming the data payload (the data channel). These protocol details have interesting effects on the LOSF problem.

3.1 Channel Establishment

GridFTP servers listen on a well-known and published port for client control channel connections. Once a client successfully forms a control channel with a server (this often involves authentication and authorization), it can begin sending commands to the server.

In order to transfer a file, the client must first establish a data channel. This involves sending the server a series of commands on the control channel describing attributes of the desired data channel such as: what protocol to use, binary or ASCII data, passive or active connection, and various protocol specific attributes. Once these commands are successfully sent, a client can request a file transfer. At this point a separate data channel connection is formed using all of the agreed-upon attributes, and the requested file is sent across it.

In standard FTP the data channel can be used only to transfer one file. Future transfers must again go through the process of setting up a new data channel. GridFTP modified this part of the protocol to allow many files to be transferred across a single data channel. With GridFTP all of the messaging to establish a data channel is done once; the data channel connection is formed just once, and the client can request several file transfers using that same data channel. This enhancement is known as *data channel caching*.

3.2 File Transfers

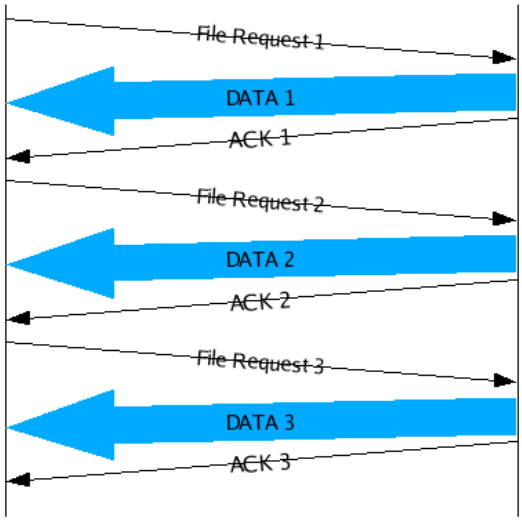

Figure 1: GridFTP file transfers with no pipelining

File transfer requests are done with the RETR (send) or STOR (receive) command. A client sends one of these commands to the server across the control channel. Data then begins to flow between the client and server over the data channel. Once all of the data has been transferred, a "226 Transfer Complete" acknowledgment message is sent from the server to the client on the control channel. Only when this acknowledgment is received can the client request another transfer. This interaction is illustrated in Figure 1.

As the figure shows, there is an entire round-trip time on the control channel between transfers where the data channel must be idle. Before issuing the next transfer command the client must first receive the transfer completion acknowledgment, which is one trip across the network. After receiving the acknowledgment, the client sends the transfer command immediately. However, the server does not immediately receive it. The message must cross the network before the server will begin sending data. This process involves another trip across the network. Assuming we have the GridFTP data channel caching enabled, we do not have to worry about the latencies involved with establishing the data channel. If we do not have it enabled, the delay is significantly longer.

During this time the data channel is idle. The latency between transfers adds to the overall transfer time and thus detracts from the overall throughput. The problem is even exacerbated when communicating over high-latency networks where the RTT is very high. While the idle data channel time is a problem, there is a far greater problem that it causes. TCP is a window-based protocol. For it to achieve maximum efficiency, the window size of allowed unacknowledged bytes must grow to the bandwidth delay product [12]. Various algorithms in the TCP protocol decide to increase or decrease the window size based on observed events [13]. If a connection is idle for longer than one RTT, the window size gets reduced to zero; and once it is used again, it must go through TCP slow start [14]. When transferring a series of files, the data channel is idle for a control channel RTT in between transfers. If the control channel RTT and the data channel RTT are similar, it is likely that data channel TCP connections will have entire closed windows by the time the next transfer begins.

When the amount of data sent in each file is small, the ratio of idle data channel time to transfer time becomes higher and affects the throughput. Additionally, small files may not be transferred long enough to traverse the slow-start algorithm and bring TCP to full throttle. Thus, even when data is being transferred, it is not moving at full speed.

4 PIPELINING

Pipelining approaches the LOSF problem by trying to minimize the amount of time between transfers. Pipelining allows the client to have many outstanding, unacknowledged transfer commands at once. Instead of being forced to wait for the "226 Transfer Successful" message; the client is free to send transfer commands at any time. The server processes these requests in the order they are sent. Acknowledgments are returned to the client in the same order. The process is shown in Figure 2.

This process hides the latency of transfer requests by overlapping them with data transfers. The first transfer request is sent, and data begins to flow across the data channel. While the file transfer is in progress, the client sends the next n file transfer requests. The server queues the requests. When the server completes the file transfer, it sends the acknowledgment to the client and checks the queue for the next transfer request. If the queue is not empty, the next file transfer begins immediately. There is some inevitable processing latency between transfers, but it is very small compared to the entire RTT of network latency that has been eliminated.

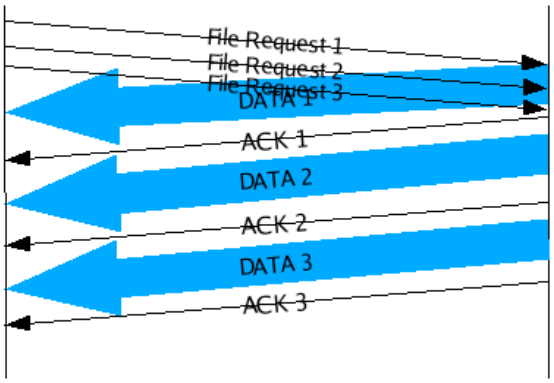

Figure 2: GridFTP file transfers with pipelining

According to the proposed pipelining protocol, the client is allowed to send an unlimited number of outstanding commands. In practice, the number of outstanding commands will be limited by the GridFTP server implementation and TCP flow control. The client is free to send as many commands as it wishes on the TCP control channel. However, the GridFTP server will read a limited number of these commands out of the TCP buffer and into its process space. All other outstanding commands will remain in the operating systems TCP buffers. As the server side buffers get full, the TCP window will close. Ultimately, the sending side TCP buffers will fill up, and the client's attempt to send future commands will be stalled. In most cases there is little performance benefit for a client to have more than three outstanding commands; however, allowing an unlimited number makes client implementation simpler.

5 IMPLEMENTATION

We implemented the pipelining solution in the Globus Toolkit GridFTP libraries and jglobus cog lava libraries. The GridFTP server was modified to read commands off of the control channel and into a queue. The maximum number of commands allowed in the queue is configurable, but the default is 20. The main reasons for setting a maximum number are to preserve main memory and to prevent potential denial of service attacks. When the server is ready to process a command, it pulls out a command from the queue and processes it. The server's thread of execution was left entirely untouched in this implementation. This fact illustrates the minimal additional processing required for our implementation.

The majority of the modifications are on the client side of the protocol exchanged. We used the jglobus [16] Java libraries as our client. We modified the API to allow a user request many transfers at one time. All of the transfer requests are immediately sent to the server, and the client waits for the same number of acknowledgments from the server.

6 RESULTS

To show the effectiveness of pipelining, we ran a series of experiments. All of our experiments were performed on TeraGrid [15] machines. For local-area tests we ran entirely on the University of Chicago TeraGrid. Our wide-area tests ran between the San Diego Supercomputer Center TeraGrid site and the University of Chicago TeraGrid site. The nodes at these sites are Dual Itanium 1.5 GHz machines with 4 MB of RAM and 1 Gb/s network interface cards. We used the Globus GridFTP server with the modifications described above and a custom client written by using the jglobus libraries described above. To avoid anomalies and bottlenecks in the filesystem, we used the standard UNIX devices /dev/zero and /dev/null as our source and desitation files, respectively. The devices appear as files to the GridFTP server; however, they do no disk or block I/O.

Figures 3 through 6 show the results of an experiment that transfers 1 GB of data partitioned into an increasing number of files. As the number of files increases, the size of each file decreases, but the total number of bytes transferred remains constant at 1 GB. The top x-axis shows the number of files, and the bottom x-axis shows the size of each file. The y-axis shows the achieved throughput in Mb/s.

The LAN results in Figures 3 and 4 show how the legacy transfer request techniques quickly suffer when the data is partitioned into multiple files. There is a significant dropoff before just 10 files of 100 MB each, and almost all of the throughput is lost at 1,000 1 MB files. However, the pipelining solution is unaffected by file partitioning until the point where the file sizes are less than 100 KB. The wide-area tests in Figures 5 and 6

show how significantly latency affects the legacy transfers. Sine the round-trip times are greater on wide area networks, the delay between transfers is also greater, and thus the overall transfer time is longer. However, the pipelining case is again unaffected.

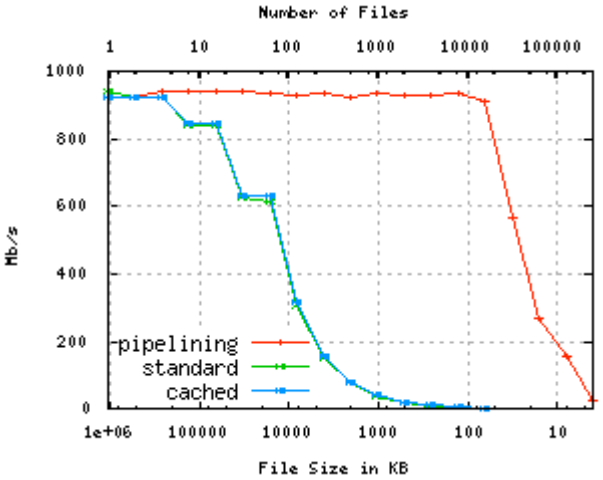

Figure 3: Comparison of the performance of pipelined GridFTP transfers with standard (nonpipelined) GridFTP transfers in a LAN with no security

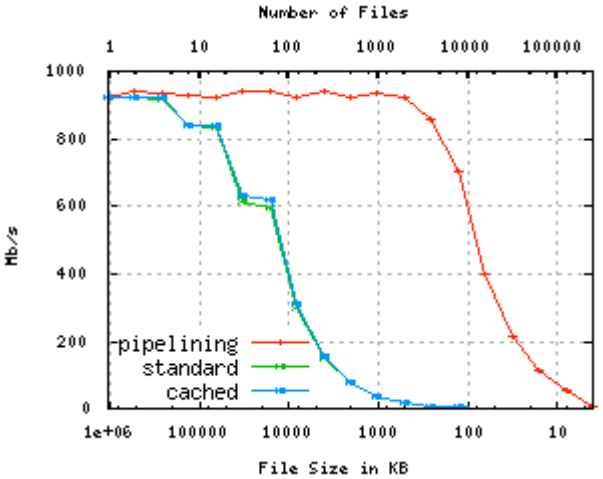

Figure 4: Comparison of the performance of pipelined GridFTP transfers with standard (nonpipelined) GridFTP transfers in a LAN with security

Security affects the results in a way we did not expect. Since we are caching data channel connections in both the *cached* and the *pipelining* cases, we did not expect the throughput levels to drop any sooner with security than without security. However, as shown in Figures 4 and 6, this is not the case. As the number of files increases, the throughput drops off sooner when sending with GSI authentication. After extensive investigation we have determined that this result is due not to any data channel handling but rather to message-processing latencies on the control channel.

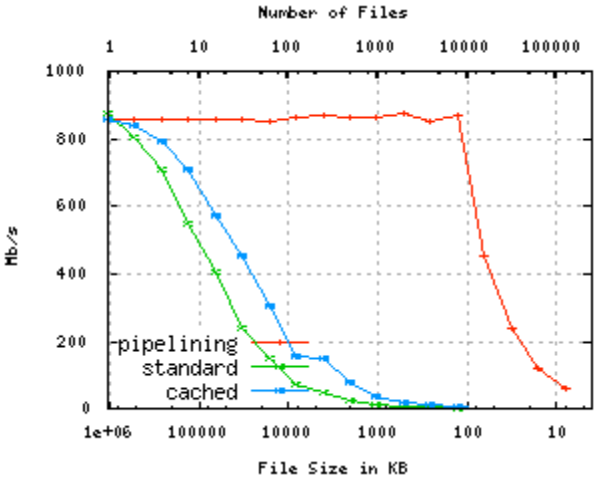

Figure 5: Comparison of the performance of pipelined GridFTP transfers with standard (nonpipelined) GridFTP transfers in a WAN with no security

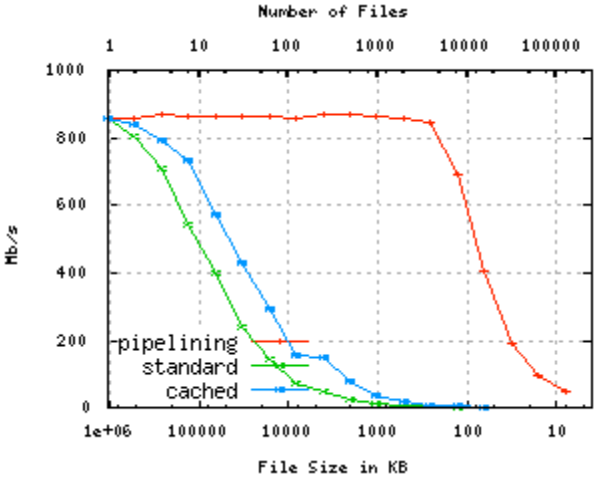

Figure 6: Comparison of the performance of pipelined GridFTP transfers with standard (nonpipelined) GridFTP transfers in a WAN with security

Between transfers the server sends a reply to the client. In our implementation the data channel must be idle while the reply is formatted and passed to the TCP stack for sending. With nonsecure transfers this time is extremely short. With GSI, however, the reply must be encrypted, and therefore it takes much longer to format. As more transfers are requested, more of these replies must be sent. Thus, this idle time becomes great enough to affect the transfer rate.

7 Conclusions and Future Work

We have presented a solution to the LOSF problem that is immediately available and implemented in a

widely deployed GridFTP server. No additional computational or data transmission work is added as part of the pipelining solution. The solution changes only the order in which the work is done.

Our results show that the pipeline approach is effective up to the point where the files are so small that the overhead for processing file opens takes effect. We plan to address this problem by adding an "open ahead" feature to the GridFTP server and by applying techniques to make better use of multicore processors for secure messaging.

Another problem we observed during this work is that a race condition exists in the Mode E [1] data channel protocol. When cached data connections are used with many parallel TCP connections, data can arrive out of order in an undetectable way. The problem occurs if the level of parallelism is adjusted between file transfers on a cached connection. Prior to the pipelining implementation it was extraordinarily unlikely that this would happen because of the high latency between transfers. With pipelining, however, the likelihood increases. We have modified the Globus implementation of mode E to avoid this problem in a safe way; however, the protocol description should be modified to eliminate this race possibility.

ACKNOWLEDGMENTS

This work was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Dept. of Energy, under Contract DE-AC02-06CH11357 and in part by SciDAC-2 CEDPS. We are grateful for having access to the TeraGrid: Without a multisite wide-area test bed we could not have run these experiments, and we would not have gained the insights that we now have for future work.

8 REFERENCES

[1] W. Allcock, "GridFTP: Protocol extensions to FTP for the Grid," Global Grid ForumGFD-R-P.020, 2003.

[2] Foster, I. and Kesselman, C. Globus: A Metacomputing Infrastructure Toolkit. International Journal of Supercomputer Applications, 11 (2). 115-128. 1997.

[3] W. Allcock, J. Bresnahan, R. Kettimuthu, M. Link, C. Dumitrescu, I. Raicu, and I. Foster, "The Globus striped GridFTP framework and server," in SC'05, ACM Press, 2005.

[4] W. Allcock, A. Chervenak, I. Foster, L. Pearlman, V. Welch, and M. Wilde, "presented at International Conference on Computing in High Energy and Nuclear Physics, Beijing, China, September 2001.

[5] N. Karonis, M. E. Papka, J. Binns, J. Bresnahan, J. A. Insley, D. Jones, and J. Link, "High-Resolution Remote Rendering of Large Datasets in a Collaborative Environ-ment," Future Generation of Computer Systems, pp. 909-917, 2003.

[6] A. Chervenak, R. Schuler, C. Kessel-man, S. Koranda, B. Moe, "Wide Area Data Replication for Scientific Collaborations," Proceedings of 6th IEEE/ACM International Workshop on Grid Computing (Grid2005), November 2005.

[7] W. Allcock, J. Bester, J. Bresnahan, A. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, and S. Tuecke, "Data management and transfer in high performance computational grid environments," Parallel Computing Journal, vol. 28, no. 5, pp. 749~771, 2002.

[8] http://www.gnu.org/software/tar/

[9] K. Bayer. "GNU Tar Streaming within GridFTP and the Globus Toolkit," Master's Thesis, Harvard University, March 2006.

[10] Mandrichenko, I. GridFTP Protocol Improvements. Global Grid Forum, GWD-E-21, 2003.

[11] Postel, J. and Reynolds, J. File Transfer Protocol. Internet Engineering Task Force, RFC 959, 1985.

[12] S. Floyd, "HighSpeed TCP for large congestion windows," RFC 3649, Experimental, December 2003.

[13] M., Paxson, V. and Stevens, W. TCP Congestion Control. IETF, RFC-2581, 1999.

[14] V. Jacobson. Congestion avoidance and control. In Proceedings of SIGCOMM '88: Communications, Architec-tures, and Protocols, pages 314--329. ACM SIGCOMM, August 1988

[15] http://www.teragrid.org/

[16] http://dev.globus.org/wiki/CoG_jglobus

License