Managed GridFTP

John Bresnahan, Michael Link, Rajkumar Kettimuthu, Ian Foster
Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL
Computation Institute, University of Chicago/Argonne National Laboratory, Chicago, IL

***Abstract*— GridFTP extends the standard FTP protocol to provide a high-performance, secure, reliable data transfer protocol optimized for high-bandwidth wide-area networks. The Globus GridFTP implementation has become the preeminent high-performance data transfer tool for the Grid community. Its modular architecture enables a standard GridFTP-compliant client access to any storage system that can implement its data storage interface, including the HPSS archival storage system, SRB, the GPFS parallel file system, and POSIX file systems. Its eXtensible I/O interface allows GridFTP to target high-performance wide-area communication protocols such as UDT, FAST TCP, and RBUDP. The Globus GridFTP server implementation already implements the concept of "striping," where multiple data movers are aggregated as one logical resource. However, there exists no mechanism in GridFTP to manage the use of server resources by the clients. When many transfer sessions occur simultaneously, the GridFTP server can overwhelm the transfer host and/or the underlying storage system. Moreover, there is no flexibility in the management of data movers in striped configuration. The data movers must be statically configured, and they cannot come and go dynamically. In this paper, we present a framework to manage the GridFTP resources efficiently so as to avoid overburdening host resources, prevent client starvation, and enable dynamic addition or removal of data movers.**

Keywords – Managed data movement, Grid data movement, high-speed transfers, secure WAN transfers

I. INTRODUCTION

The amount of data that science experiments and simulations produce keeps increasing, and the need to transport these data over wide-area networks is also growing, as many science experiments are collaborative and involve scientists who are geographically dispersed. The GridFTP protocol [1] was defined to make the transport of data secure, reliable, and efficient for these distributed science collaborations. The GridFTP protocol extends the standard File Transfer Protocol (FTP) [2] with useful features such as Grid Security Infrastructure (GSI) security [3], increased reliability via restart markers, high-performance data transfer using striping and parallel streams, and support for third-party transfer between GridFTP servers.

The Globus GridFTP implementation [4] has become the preeminent high-performance data transfer tool for the Grid community. Its modular architecture enables a standard GridFTP-compliant client access to any storage system that can implement its data storage interface [5], including the HPSS archival storage system [6], SRB [7], the PVFS parallel file system [8], the GPFS parallel file system [9], and POSIX [10] file systems. Its eXtensible I/O interface [11] allows GridFTP to target high-performance wide-area communication protocols such as UDT [12], FAST TCP [13], and RBUDP [14]. Globus GridFTP is optimized to handle various types of datasets—from a single, huge file to datasets comprising lots of small files [15, 16].

Even though GridFTP has been widely adopted and has proven to be a robust tool for bulk data movement over wide-area networks, there remains significant room for improvement in terms of managing the GridFTP server's underlying resources, such as the host CPU and memory, file systems and the data mover processes.

GridFTP is a high-performance data transport protocol. In order to provide the fastest data transfer when using TCP-based data channels, GridFTP allows the client to set the TCP buffer size [17]. This value directly affects the amount of memory a given transfer session will attempt to acquire. A greedy client could overprovision itself and monopolize a precious resource, negatively impacting both the performance and the stability of the entire system.

Further, when simultaneous GridFTP transfer sessions compete for system memory, such competition can result in thrashing and other situations that cause suboptimal performance. Often, if a transfer session is stalled until more resources are available, the overall throughput is higher than when resources are split among too many simultaneous transfer sessions. In extreme cases the available memory can be entirely exhausted. Many systems attempt to recover from this situation by running the OOM (out-of-memory) handler. This will kill a non-determinate process, causing unknown effects on the system. Further, the system can be so heavily overloaded that system administrators cannot even SSH into the machine and thus cannot manually free resources.

GridFTP can be run as a striped server [4] where there are several data movers and a control node. The data movers run in tandem to transfer files faster by tying together many NICs. The control node is the client contact point where transfer requests are made. There is a limitation that the list of possible data movers must be statically configured. Unfortunately, data movers tend to come and go. Sometimes data movers fail, and sometimes more data movers need to be added to the pool. Not only are data movers and control processes separate processes, but for the sake of security each GridFTP user connection to the GridFTP server is sandboxed into a separate process. This approach results in many separate UNIX processes running in tandem. We note that although they are viewed as separate processes to the involved operating systems, they are all working together as part of the GridFTP server

and the resources they acquire must be managed as such.

We present here a framework that enables the GridFTP service to share state across client connections and thus manage its resources efficiently so as to avoid overwhelming host resources, prevent client starvation, and enable dynamic addition or removal of data movers. The framework includes Globus Fork (GFork), a user-configurable super-server daemon similar to xinetd [18] that enables sharing of state across client connections for a service, and user-defined master programs that coordinate resource sharing. Using a simple memory management algorithm, we demonstrate how this framework can effectively manage the memory usage for GridFTP client requests to prevent system meltdown. Further, we illustrate how this framework can be used to dynamically add or remove GridFTP resources

The rest of the paper is as follows. In Section II, we provide the motivation for developing this framework. We present the framework in Section III. In Section IV, we evaluate a simple memory management scheme for GridFTP using this framework. In Section V, we showcase how this framework can be used to change the GridFTP resources based on demand. We summarize our work and discuss future work in Section VI.

II. MOTIVATION

Open Science Grid [19] participants reported that their single biggest problem with running GridFTP servers is that they can overwhelm the transfer host and/or the underlying storage system. To access the server, a user must be authenticated, have appropriate read and write permissions, and respect the total connection limit; but beyond these requirements, there is no management or control. A user can hold a connection open indefinitely and move an unlimited number of files (barring disk space or system quota constraints). A more flexible management clearly is needed to limit the length of time a user can hold a connection, address prioritization and responses to overburdened services, and prevent starvation.

Sites with one or more 10 Gbs links are becoming commonplace in scientific environments. It can take substantial resources at the end points to fully utilize such a connection, either because the end hosts have only a 1 Gbs NIC or because multiple hosts are needed to get the required data rate from the storage subsystem. If dedicated transfer resources are used, a major investment is required. The ability to dynamically provision transfer resources will be of significant benefit to any large installation that wishes to provision for normal load and then dynamically allocate additional resources for peak loads.

For example, the striped GridFTP services in TeraGrid's [20] current production operations typically do not include enough server nodes to fill TeraGrid's 10 Gb/s WAN links. It isn't clear that adding additional dedicated server nodes would be justified by demand. On the other hand, it seems likely that some transfer requests would benefit significantly from additional server nodes. Hence, it would be good if there were a way for additional nodes to be available on demand to satisfy these requests.

III. GFORK

GFork is a user-configurable super-server daemon similar to xinetd in that it listens on a TCP port. When clients connect to a port, GFork runs an administrator-defined program that services that client connection, just as xinetd does.

Xinetd is a time-tested service container that sandboxes client connections securely into user-level local processes. This greatly limits the security risks of a service and the damage potential of bugs. A drawback to xinetd, however, is that there is no way to maintain or share long-term information. Every time a client connects, a new process is created; and every time that client disconnects, the process is destroyed. All of the information regarding the specific interactions with a given client is lost with these transient processes. A further disadvantage is that there is no way for these service instances to share service-specific information with each other while they are running.

Sometimes it is useful for a service to maintain long-term service-specific state or for a service to share state across client connections. GFork is designed to address this situation. As shown in Figure 1, GFork runs a long-term master program and forms communication links via UNIX pipes between this process and all client connection child processes. This configuration allows long-term state to be maintained in memory and allows for communication between all nodes.

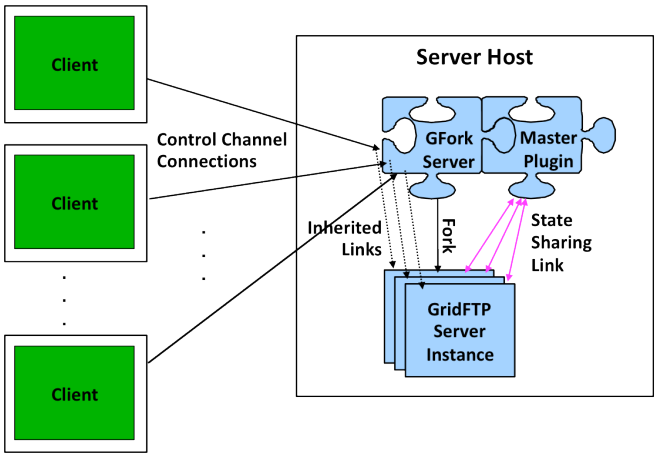

Figure 1: GFork architecture

Associated with a GFork instance is a master process. When GFork starts, it runs a user-defined master program and opens up bidirectional pipes to it. The master program runs for the lifetime of the GFork daemon. The master is free to do whatever it wants; it is a user-defined program. Some master programs listen on alternative TCP

connections to have state remotely injected. Others monitor system resources, such as memory, in order to best share resources. As clients connect to the TCP listener, child processes are forked that then service the client connection. Bidirectional pipes are opened up to the child processes as well. These pipes allow for communication between the master program and all child processes. The master program and the child programs have their own protocol for information exchange over these links. GFork is just a framework for safely and quickly creating these links.

IV. Memory Management

In order to move data at high speeds, network bandwidth must be consumed, and along with it so must disk and system bus bandwidth and, most important, main memory. To use TCP most efficiently, GridFTP allows the client to set the TCP buffer size [17]. The TCP buffer size directly affects the amount of memory a given transfer session will attempt to acquire.

Because the client controls the amount of potential memory that its session will require and because of a common memory-provisioning model known as optimistic provisioning, it is possible that under heavy loads a GridFTP server can consume all the system's memory resources. Optimistic memory allocation means that the kernel is willing to allocate more virtual memory than there is physical memory, based on the assumption that a program may not need to use all the memory it asks for. When all of the physical memory is used, the kernel's attempt to map more virtual memory to physical memory will cause it to enter an "out-of-memory" (OOM) condition. This will trigger the OOM handler, which kills processes on the system in order to reclaim resources. Unfortunately, it is difficult to control or determine which processes the OOM handler will kill; therefore it is difficult to automatically recover form an OOM exception in a non-disruptive way.

Even if the OOM handler is not tripped, overprovisioning memory for TCP buffers can cause suboptimal situations. If memory pages are swapped in and out too often, the system can thrash so heavily that an administrator cannot acquire a working terminal (either remotely via SSH or even on the console). In the worst case the only solution is a reboot.

Further, not just the abuse of system memory and OS schedulers can cause problems: TCP itself can be put in a suboptimal state if too many connections are too aggressive with their window sizes. Instead of having all connections with smaller, yet fully open windows, the results are very large potential windows, but in a constant state of AIMD [21] due to packet loss.

GFork can help avoid this situation through a master plug-in that manages memory across GridFTP sessions and makes sure the system is not overloaded. Our focus is on the impact of a memory-management GFork plug-in on the GridFTP throughput and resource usage. The plug-in controls the amount of memory a transfer session can use in order to avoid a critically low amount of memory and a misuse of other system resources.

GridFTP uses memory in three main areas:

1) *Application*
   In order to fork a new instance of GridFTP that will service the client connections, a certain amount of memory is required. This is static and independent of a client request.
2) *I/O Buffers*
   This is the memory buffer to which data is read from the network/disk, and then from which data is written to the disk/network.
3) *Kernel Memory*
   This is the main culprit. When the client requests a TCP buffer size of *x* MB, the operating system (for example, Linux) that does optimistic allocation accepts the request (up to a sysctl.conf configured maximum) but does not reserve any memory. This approach leads to problems because the operating system can overprovision itself. The TCP flow control algorithms believe they have the requested amount of memory to work with, but the operating system may be writing checks that it cannot cash.

A typical approach for memory management on the GridFTP server nodes is to limit the number of clients that are allowed to form a TCP connection. When a TCP connection is formed, inetd or xinetd executes a GridFTP server process. A GridFTP transfer session requires 2 MB of memory to be at all useful. Therefore, the recommended limit on the number of allowed connections is |RAM| / 2 MB, up to a maximum of 100 simultaneous connections.

Limiting the number of simultaneous transfer sessions is basic and doesn't allow for optimal memory usage. It is simply the first phase to prevent obvious system overloads. GFork allows for better memory and connection management. The GFork memory management plug-in keeps track of the number of session transfers. When a new session begins, the plug-in sends it a message allowing it to use a certain amount of memory (the amount is determined by an algorithm described below). That amount is subtracted from a count representing the total available memory. When the session ends, the given memory is added back to the total.

A transfer session uses two I/O buffers for every TCP stream that it has. The size of the I/O buffer is the same as the size of the requested TCP buffer size. For this reason we limit the maximum size of any transfer session to one-third the amount of memory that the plug-in gives it.

If the plug-in has less than 2 MB of available memory, no new transfer sessions are allowed to be created. This limitation is different from the original connection limitation in that it is based on the amount of used memory, not a static value based on total system memory.

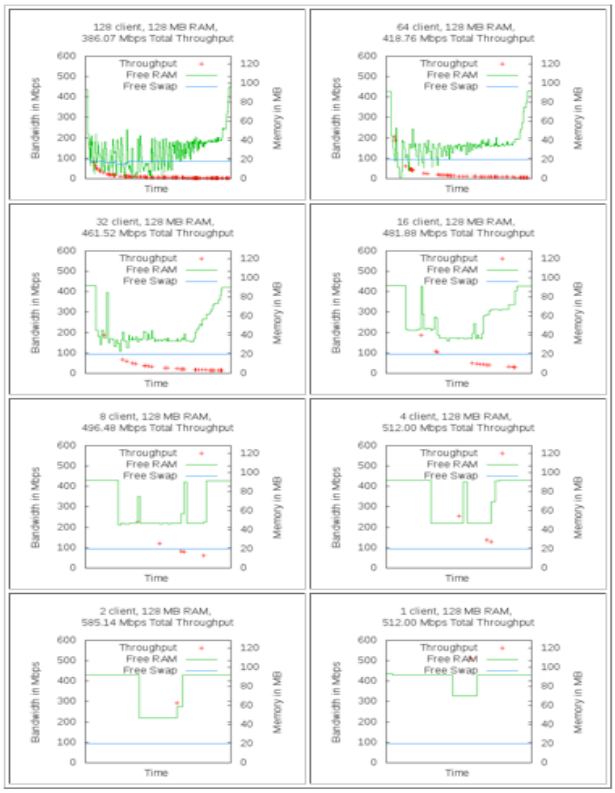

Figure 2: Memory usage and throughput characteristics on a 128 MB machine for varying number of concurrent clients each using a single TCP stream – with memory limiting

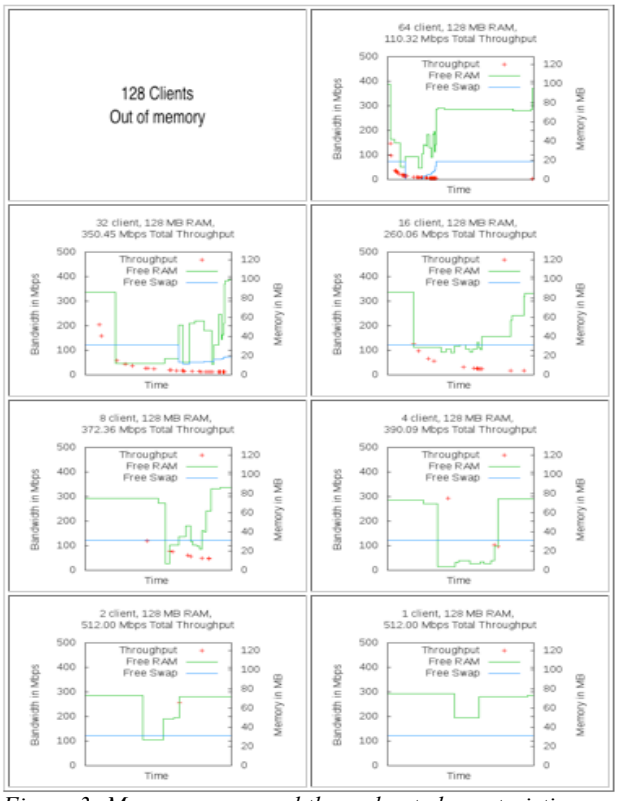

Figure 3: Memory usage and throughput characteristics on a 128 MB machine for varying number of concurrent clients each using a single TCP stream – with no memory limiting

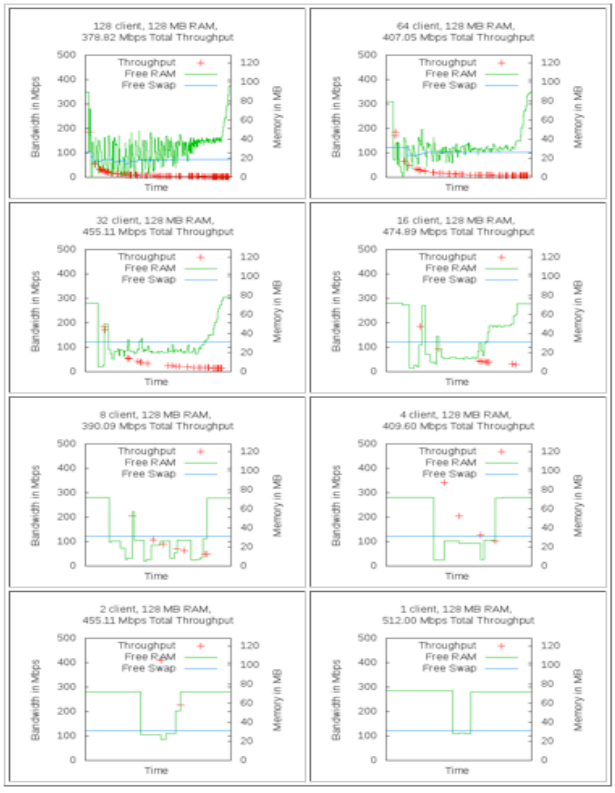

Figure 4: Memory usage and throughput characteristics on a 128 MB machine for varying number of concurrent clients each using 2 parallel TCP streams – with memory limiting

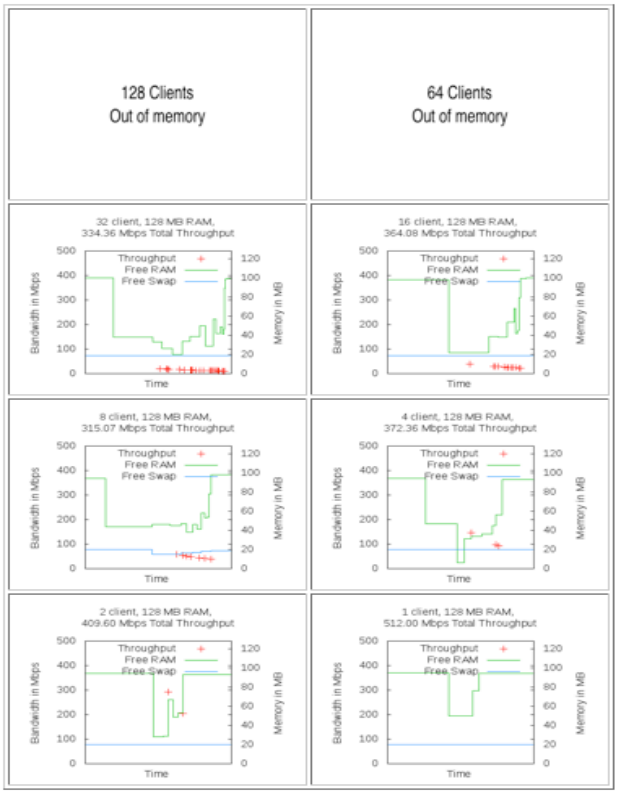

Figure 5: Memory usage and throughput characteristics on a 128MB machine for varying number of concurrent clients each using 2 parallel TCP streams – with no memory limiting

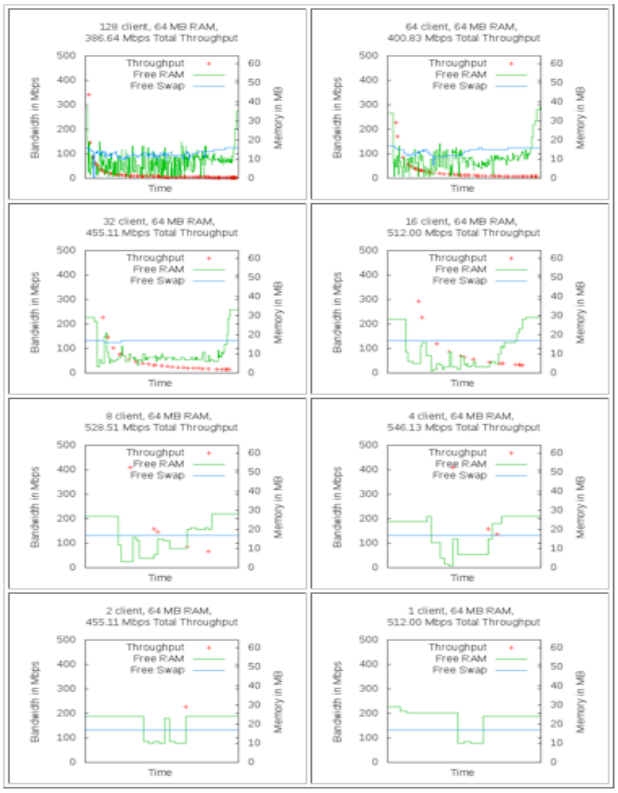

Figure 6: Memory usage and throughput characteristics on a 64 MB machine for varying number of concurrent clients each using a single TCP stream – with memory limiting

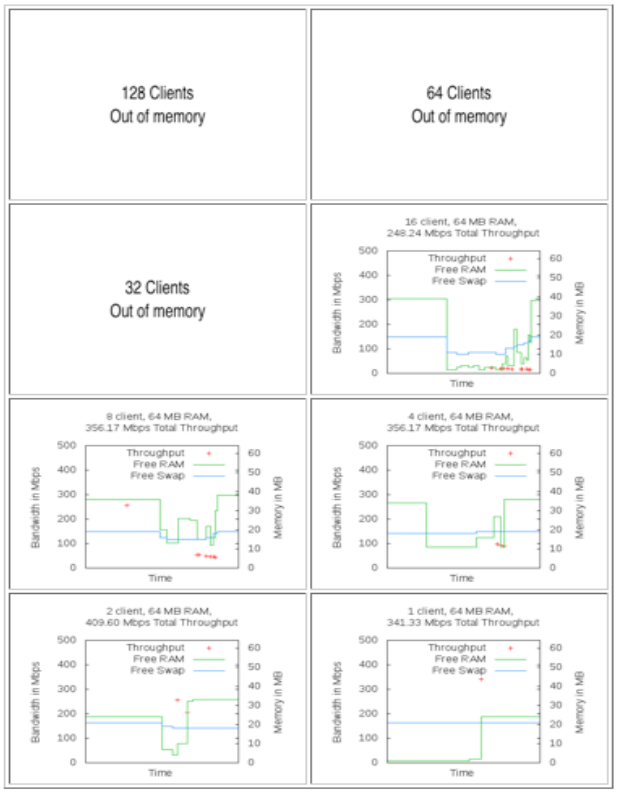

Figure 7: Memory usage and throughput characteristics on a 64 MB machine for varying number of concurrent clients each using a single TCP stream – with no memory limiting

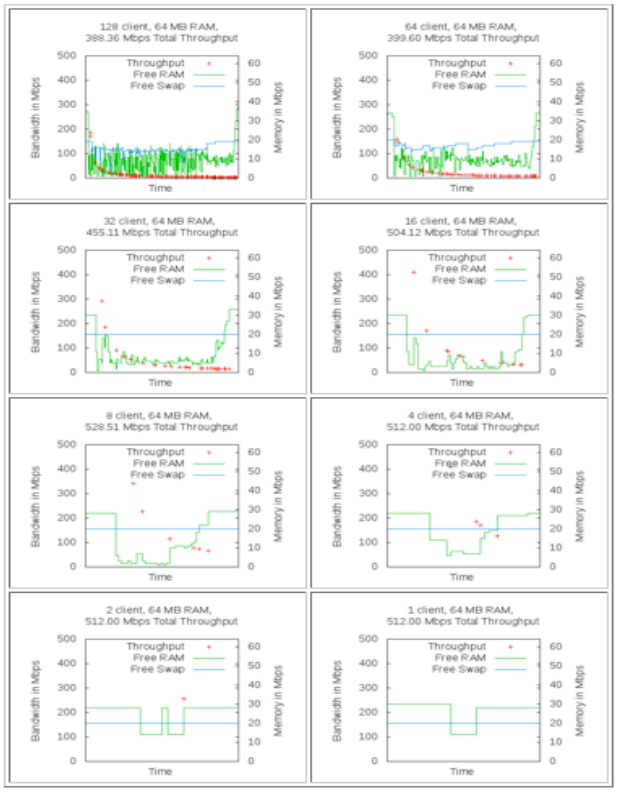

Figure 8: Memory usage and throughput characteristics on a 64 MB machine for varying number of concurrent clients each using 2 parallel TCP streams – with memory limiting

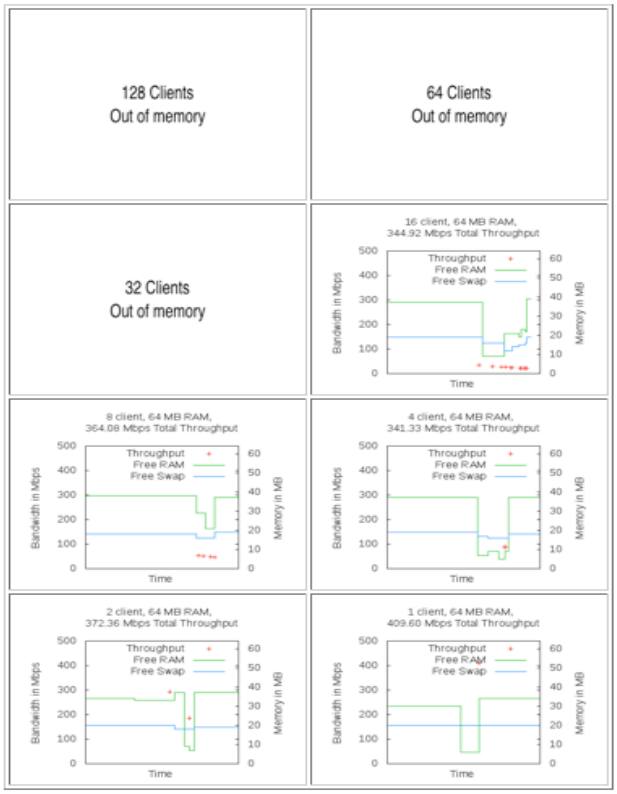

Figure 9: Memory usage and throughput characteristics on a 64 MB machine for varying number of concurrent clients each using 2 parallel TCP streams – with no memory limiting

A. *Memory Management Algorithm*

We evenly divide 90% of the memory to the first 10% of the allowed connections. Thus, on a system with 2 GB of RAM, which allows 100 simultaneous connections for the first 10 connections, will have 180 MB with which they can work. After the first 10% of the connections are formed, each future connection receives half of what is available, down to a threshold where no future connections are allowed. Each of the first 10 connections gets 9% of the total available memory. The 11th connection gets 5%, the 12th connection 2.5%, the 13th 1.25%, and so forth until the memory allocated remains greater than or equal to 2 MB. Let us say there are 13 simultaneous connections to start with. Then the first 4 of these 13 connections finish, and a new connection (14^{th} connection) request arrives. The memory management plug-in does not change the allocation for the connections that are already active. Now, since there are only 9 active connections, the new connection request will get 9%, but the three connections that arrived just before this connection will have less memory. The allocation is not totally fair; but since it ensures that each connection gets enough memory (2 MB) to make some progress, any one connection does not get penalized too much.

B. *Experimental Results*

To measure the effectiveness of the memory limiting system, we needed an environment with very low memory and configurable amounts of memory. We achieved this with a Xen virtual machine (VM) [22]. In all our experiments, clients were run on a 2 GHz processor with 512 MB of RAM, and the server was run in a Xen VM with either 64 MB or 128 MB of RAM (depending on the experiment) with a 2 GHz AMD 64 bit processor. The machines were connected to each other via a 1 Gb/s network with Iperf [23] measured maximum speeds of 600 Mb/s.

In the experiments we ran an increasing number of simultaneous clients requesting transfers of 256 MB files (/dev/zero to /dev/null). To limit the number of variables in the system, we did transfers from /dev/zero to /dev/null. This approach allowed us to focus the experiments on system memory and network transfer by removing the disk I/O variable. We measured available memory throughout the lifespan of all transfers with free.

Figures 2–9 show the results. The x-axis is increasing time throughout the lifespan of the experiments. The green line shows the amount of free RAM according to the y-axis on the right side of the graph. The blue line shows similar values for free swap. The red points show when each transfer completed and the throughput it achieved. Collective throughput of all the transfers is shown in the title of each graph. In some cases where memory was not limited, we received OOM exceptions. In these cases we display no results, but the fact that they occur in the non-limiting cases shows that our resource management scheme is successful in preventing meltdown.

Even though the total amount of memory used with the memory-limiting plug-in is sometimes more than what is used in the standard server (without any memory limiting), the memory use is well under control and it never runs into an OOM situation. Moreover, the total throughput obtained is significantly higher with increasing number of clients. A server with no memory limiting starts to eat up the swap space early on. Also, as the number of clients increases, the total throughput goes down much faster. Without any control on the use of memory for each transfer session, some transfers may take a significantly long time to finish. As Figure 3 shows, in the case of 64 simultaneous clients, one transfer finishes at the tail end of the x-axis, whereas all other transfers finish much earlier. When there are many simultaneous clients, each client may get a limited amount of memory, and thus all the clients take longer to finish. A similar situation happens with 16 simultaneous clients on a 64 MB machine (see Figure 7). These problems do not occur when the memory-limiting plug-in is in place.

V. Dynamic Provisioning of GridFTP Data Movers

GridFTP offers a powerful enhancement called striped servers. In this mode a GridFTP server is set up with a single control node and one or more data mover nodes. All of the data movers work in concert to transfer a single file and thereby achieve high throughput rates. When the frontend is run out of xinetd, the list of possible data movers must be statically configured.

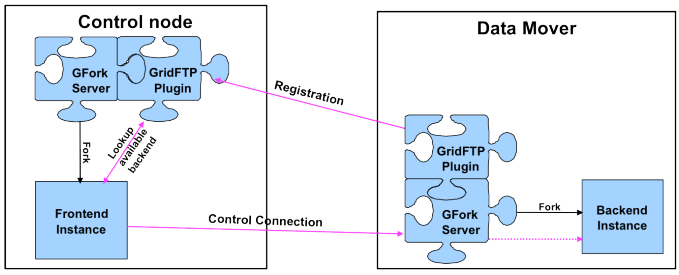

Figure 10: Dynamic data mover registration

The GFork framework allows for dynamic addition and removal of data mover nodes. Figure 10 illustrates the interaction between the control and data mover nodes. Both the control node and the data mover node have a GFork master plug-in. The GFork master program on the control node listens on a user-configurable port for data mover registrations. The GFork master program on the data movers should be configured with the control node's registration contact point. Data movers can then connect to the control node to notify their existence. By default, a registration is good for 10 minutes, but a data mover is free

to refresh its registration. The GFork master program on the control node can be configured with a list of authorized distinguished names (DNs) [24]. In order to register, the backend must authenticate and provide its DN. The provided DN is checked against the list of authorized DNs, and the registration is allowed only if the data mover's DN is in the authorized DN list. The master program can be configured to have no registration security at all.

The control node master plugin configuration allows the administrator to specify the number of data movers to use for each transfer session. When the control node gets a transfer request it picks the specified number of data movers from the list of available data movers in a round-robin fashion to service the transfer request. If the load on the existing data movers goes above a certain threshold, administrators can easily add more data movers to handle the demand without any disruption to the service. Similarly, they can shut down data movers when the demand is less.

VI. Summary and Future work

We have developed a resource management framework for GridFTP. The framework was motivated by the need to manage the GridFTP resources effectively and rotect from any unintentional or intentional misuse. To this end, we created a simple memory management plug-in and showed the effectiveness of this framework in preventing system meltdown. To facilitate dynamic addition and removal of GridFTP data movers, we implemented secure registration capabilities using this framework.

In future, we intend to add more sophisticated resource management capabilities to provide better than best-effort data movement capabilities in GridFTP. GFork in the present form runs only one service per instance; it takes a single configuration file and handles a single service. Thus, it differs from xinetd, which runs many services per instance, all associated with many different ports. We plan to enhance GFork to handle many services in the way that xinetd does.

Acknowledgment

This work was supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

References

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

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

[3] I. Foster, C. Kesselman, G. Tsudik, S. Tuecke, "A Security Architecture for Computational Grids," in *5th ACM Conference on Computer and Communications Security Conference*, 1998, pp. 83-92.

[4] W. Allcock, J. Bresnahan, R. Kettimuthu, M. Link, C. Dumitrescu, I. Raicu, and I. Foster, "The Globus Striped GridFTP Framework and Server," SC'05, ACM Press, 2005

[5] R. Kettimuthu, M. Link, J. Bresnahan, W. Allcock, "Globus Data Storage Interface (DSI) - Enabling Easy Access to Grid Datasets," 1st DIALOGUE Workshop: Applications-Driven Issues in Data Grids, Aug. 2005.

[6] Watson, R.W. and Coyne, R.A. The Parallel I/O Architecture of the High-Performance Storage System (HPSS). IEEE MSS Symposium, 1995.

[7] Baru, C., Moore, R., Rajasekar, A. and Wan, M., The SDSC Storage Resource Broker. 8th Annual IBM Centers for Advanced Studies Conference, Toronto, Canada, 1998.

[8] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur, "PVFS: A Parallel File System For Linux Clusters", Proceedings of the 4th Annual Linux Showcase and Conference, Atlanta, GA, October 2000

[9] General Parallel File System (GPFS), 2004. www-1.ibm.com/servers/eserver/clusters/software/gpfs.html.

[10] POSIX 1003.1e draft specification "http://www.suse.de/~agruen/acl/posix/posix_1003.1e-990310.pdf"

[11] Allcock, W., Bresnahan, J., Kettimuthu, R. and Link, J., The Globus eXtensible Input/Output System (XIO): A Protocol-Independent I/O System for the Grid. *Joint Workshop on High-Performance Grid Computing and High-Level Parallel Programming Models held in conjunction with International Parallel and Distributed Processing Symposium*, 2005.

[12] Gu, Y. and Grossman, R.L., UDT: An Application Level Transport Protocol for Grid Computing. *Second International Workshop on Protocols for Fast Long-Distance Networks*, 2003.

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

[14] He, E., Leigh, J., Yu, O. and DeFanti, T.A., Reliable Blast UDP: Predictable High Performance Bulk Data Transfer. *IEEE Cluster Computing*, 2002.

[15] J. Bresnahan, M. Link, R. Kettimuthu, D. Fraser, and I. Foster, "GridFTP Pipelining," in *Teragrid 2007 Conference* Madison, WI, 2007.

[16] R. Kettimuthu, A. Sim, D. Gunter, B. Allcock, P. Bremer, J. Bresnahan, A. Cherry, L. Childers, E. Dart, I. Foster, K. Harms, J. Hick, J. Lee, M. Link, J. Long, K. Miller, V. Natarajan, V. Pascucci, K. Raffenetti, D. Ressman, D. Williams, L. Wilson, L. Winkler, *"Lessons learned from moving Earth System Grid data sets over a 20 Gbps wide-area network"*, 19th ACM International Symposium on High Performance Distributed Computing (HPDC), 2010
Transactions on Computer Systems, *6* (1). 51-81.
1988.

[17] TCP-tuning http://www.psc.edu/networking/projects/tcptune/

[18] http://www.xinetd.org/

[19] Open Science Grid http://www.opensciencegrid.org/

[20] Catlett, C. The TeraGrid: A Primer, 2002. www.teragrid.org.

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

[22] http://www.xen.org/

[23] http://sourceforge.net/projects/iperf/

[24] S. Tuecke, V. Welch, D. Engert, L. Pearlman, and M. Thompson, "Internet X.509 Public Key Infrastructure (PKI) Proxy Certificate Profile," IETF, RFC 3820, 2004