# A Data Transfer Framework for Large-Scale Science Experiments

Wantao Liu[1,5], Brian Tieman[3], Rajkumar Kettimuthu[4,5], Ian Foster[2,4,5]
[1]School of Computer Science and Engineering, Beihang University, Beijing, China
[2]Department of Computer Science, The University of Chicago, Chicago, IL
[3]Advanced Photon Source, Argonne National Laboratory, Argonne, IL
[4]Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL
[5]Computation Institute, The University of Chicago, Chicago, IL

liuwt@act.buaa.edu.cn, tieman@aps.anl.gov, kettimut@mcs.anl.gov, foster@mcs.anl.gov

## ABSTRACT
Modern scientific experiments can generate hundreds of gigabytes to terabytes or even petabytes of data that may furthermore be maintained in large numbers of relatively small files. Frequently, this data must be disseminated to remote collaborators or computational centers for data analysis. Moving this data with high performance and strong robustness and providing a simple interface for users are challenging tasks. We present a data transfer framework comprising a high-performance data transfer library based on GridFTP, a data scheduler, and a graphical user interface that allows users to transfer their data easily, reliably, and securely. This system incorporates automatic tuning mechanisms to select at runtime the number of concurrent threads to be used for transfers. Also included are restart mechanisms capable of dealing with client, network, and server failures. Experimental results indicate that our data transfer system can significantly improve data transfer performance and can recover well from failures.

## Categories and Subject Descriptors
C.2.4 [**Distributed Systems**]: Distributed Applications

## General Terms
Algorithms, Performance, Design.

## Keywords
Data transfer, GridFTP, Data scheduling, Concurrent, Error recovery

## 1. INTRODUCTION
Modern scientific experiments and facilities, such as CERN [1], LIGO [2], the Advanced Photon Source [3], and the Spallation Neutron Source [4] can generate multiple gigabytes to terabytes of data every day. Frequently, this data must be disseminated to remote collaborators or advanced computational centers capable of running the complex CPU-intensive applications needed to analyze the data. Transferring large volumes of data on physical media such as removable disk drives is problematic. Physical media can be lost or irreparably damaged in transit. Moreover, collaborations often require access to the most current data from multiple sites around the world. Shipping data on physical media introduces a time lag and makes it difficult to ensure that all collaborators have the most recent results.

The Internet provides a convenient connection between remotely located collaborators to work on common datasets. However, transferring the large volumes of data generated by a facility or experiment over the Internet has its own set of challenges. In particular, low bandwidth and unreliable connections can make it difficult to move data rapidly and reliably.

The GridFTP [5][6] protocol extends standard FTP [7] for high-performance operation, providing improved performance compared to standard FTP. GridFTP is widely used for wide-area data transfer. For example, the high energy physics community bases its entire tiered data movement infrastructure for the Large Hadron Collider computing Grid on GridFTP; and the Laser Interferometer Gravitational Wave Observatory routinely uses GridFTP to move one terabyte a day to each of eight remote sites.

Even so, the configuration and tuning of a GridFTP toolset can be daunting to users. Errors and interruptions during data transfers are inevitable obstacles as well. Scientists desire a high-performance, straightforward, user-friendly, and robust data transfer mechanism that can significantly improve their work efficiency.

To meet this need, we have designed and implemented a data transfer framework based on GridFTP.

This paper makes four contributions: (a) a data transfer framework architecture that addresses the requirements just listed; (b) an algorithm to autotune data transfer concurrency that can improve performance significantly; (c) two data scheduling algorithms; and (d) an error recovery algorithm that addresses both client-side and server-side errors.

The paper is organized as follows. In Section 2, we review some previous work. In Sections 3 and 4, we present the data transfer framework and introduce an application of our system as a case study. In Section 5, we present experiment results; and in Section 6, we conclude and outline future plans.

## 2. RELATED WORK
Some large-scale science experiments or research projects have their own data management solution to meet their requirements. The PhEDEx [8][9] data transfer management system is used by


*DIDC'10*, June 22, 2010, Chicago, IL, USA.

the CMS experiment at CERN. PhEDEx consists of a set of agents responsible for file replication, routing decisions, tape migrations, and so on. ATLAS DDM [26] is a file-based data management system operates on the worldwide LHC computing grid infrastructure (WLCG). It is capable of managing data on the order of petabytes. Moreover, DDM has a subscription mechanism which helps user easily get the latest version of data they are interested in. The caGrid [11] aims at building a Grid software infrastructure for multi-institutional data sharing and analysis for cancer research. It has two components related to data management. caGrid Transfer [12] is used for moving small amounts of data between a client and server. For moving huge amounts of data, caGrid developed BulkData Transfer [13] based on GridFTP. Since the focus of caGrid is not on moving huge volumes of data, these transfer mechanisms offer only basic data transfer functionality, whereas our framework provides flexible data-scheduling policies and error recovery mechanisms that deals with client, network, and server errors.

Sinnott et al. [10] discuss how to manage hundreds of thousands of files produced by the nanoCMOS project. They compared the Storage Resource Broker (SRB) and Andrew File System (AFS) in terms of architecture, performance, and security. To facilitate the discovery, access, and use of electronics simulation data, they also propose a metadata management architecture. This architecture uses the SRB or AFS for data movement but does not consider error recovery and data scheduling. The work focuses on data sizes of a few gigabytes; however, we are interested in data sizes of hundreds of gigabytes or more.

Stork [14][15] is a data scheduler specialized for data placement and data movement. It is able to queue, schedule, monitor, and manage data placement activities, with data placement jobs executed according to a specified policy. Stork supports multiple data transfer protocols and can decide which protocol to use at runtime. It also implements basic error recovery mechanism. However, it cannot recover from client failures, whereas our system can.

Ali and Lauria [16] describe asynchronous primitives for remote I/O in Grid environments. The authors implemented a system, named SEMPLAR, based on the Storage Resource Broker. In addition to asynchronous primitives, multithreaded transfer and on-the-fly data compression are used to improve performance further. We also use asynchronous I/O and multithreaded transfers in our data transfer framework; in addition, however, our thread pool is able to tune dynamically at runtime to improve performance.

RFT (Reliable Transfer Service) [19] is a component of the Globus Toolkit. Implemented as a set of web services, RFT performs third-party transfers using GridFTP with basic reliable mechanisms. Data transfer state is recorded in a database; when a transfer fails, it can be restarted automatically by using the persistent data. However, our system supports not only third-party transfers but also client-server transfers. Moreover, RFT is heavyweight, relying on a database for error recovery, whereas we use a simpler and more lightweight file-based approach. In addition, RFT does not support data transfer scheduling, whereas our system supports a flexible data transfer scheduling scheme.

gLite File Transfer Service [27] provides reliable file movement in gLite grid middleware. It uses a third party copy (e.g., gsiftp) to perform the actual data movement. The transfers managed by FTS are all asynchronous. A web service interface is exposed to users. FTS has a data scheduler component as well; besides the global policy, each VO can apply their own data scheduling policies.

Vazhkudai [21] studied data replica selection, data transfer performance prediction, and parallel download of datasets from multiple servers in a data Grid environment based on Globus. Vazhkudai's work aims to improve data transfer performance by making full use of data replicas. Our work is complementary to his work. We focus on how to transfer data with high performance and robustness in an environment without replicas, since data produced by an experiment must be moved from a scientific facility to a researcher's home institute;

Using multiple streams for a data transfer can improve throughput significantly. Several researchers have sought to compute the optimal number of streams for a data transfer. Hacker et al. [22] give the relationship among throughput, number of streams, packet loss rate, and round-trip time; however, their results are valid only for uncongested networks. Lu et al. [23] and Yildirim et al. [24] extend the model to both uncongested and congested networks and present formulas for predicting the optimal number of streams. All these studies aim to optimize a single, large transfer. In contrast, we propose an effective method for tuning the throughput of multiple concurrent transfers of small files.

# 3. DATA TRANSFER FRAMEWORK

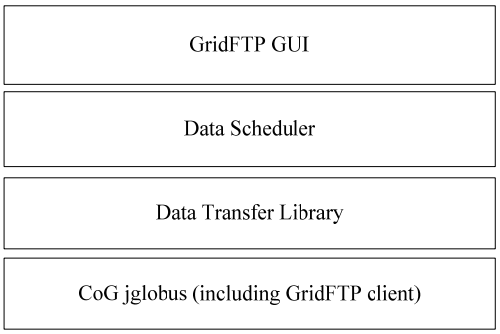


**Figure 1: Architecture of the data transfer framework**

Figure 1 shows the architecture of our data transfer framework. We use GridFTP for data movement because of its high performance and wide acceptance in the scientific community.

GridFTP GUI provides users a convenient tool for data movement based on a graphical interface. The data scheduler accepts jobs and dispatches them to the data transfer library according to a specified scheduling policy. The data transfer library hides the complexity and heterogeneity of underlying data transfer protocol. It provides a data transfer thread pool and supports error recovery. It can interact with diverse data transfer protocols, although currently we support only GridFTP using CoG jglobus. The CoG jglobus [17] library includes a pure Java GridFTP client API; it can be used to build applications that communicate with GridFTP servers.

In the sections that follow, we describe the various components of this architecture.

## 3.1 Data Transfer Library

The data transfer library (DTL) provides a simple API for asynchronous, fault-tolerant, high-performance data transfers. It accepts transfer requests from the upper layer application and

manages the data movement. The DTL is designed to be modular and extensible: diverse data transfer protocols can be easily incorporated into DTL as plugins. Currently, DTL supports only GridFTP. Other data transfer protocol plugins will be implemented in the future. DTL is not tightly coupled to the data transfer framework presented here; it is generic enough to be used separately.

Asynchronous data transfer is an efficient way to improve application performance. For example, it allows disk I/O to be overlapped with network I/O, improving resource utilization and reducing application runtime.

DTL uses thread and queue mechanisms to implement asynchronous data transfers. Transfer requests are categorized as either file requests (FRs) or directory requests (DRs), and we maintain two queues: one for file transfer requests (FQs) and one for directory transfer requests (DQs). A single directory transfer request results in a number of file transfer requests. A single thread processes the directory transfer requests in the directory queue and populates the file transfer queue. Each file transfer request is assigned a unique identifier. Each queue has a tunable, maximum-length threshold; if this threshold is exceeded, a request to add transfers blocks until there is enough space in the queue. In order to make full use of network bandwidth, a thread pool is created to process requests in the file transfer queue. By default, the initial size of the thread pool is set to four. If FQ is empty, the corresponding processing threads are suspended until a new request is received.

Figure 2 depicts the interaction between the thread pool and queues in DTL. The directory request-processing thread acquires a DR, communicates with the specified source machine (a remote GridFTP server or the machine where DTL is running) of the request to determine the names of all regular files within the specified directory, constructs an FR for each file, and adds the new FR into FQ. The file transfer request process thread in the pool repeatedly gets an FR from FQ and performs the actual data transfer. After the transfer completes, the thread starts serving another request from the queue.

After adding a request to the corresponding queue, the invoker (the application invokes DTL directly or uses DTL through data scheduler) returns immediately and continues running other tasks without waiting for the data transfer to finish. To notify the invoker of the updated transfer status and statistics information of the request, we implemented a notification mechanism. When the transfer status changes, DTL generates a notification message and sends it to the invoker. A notification message consists of the names of the files being moved, amount of bytes transferred in this request, number of successful requests, number of failed requests, and number of remaining requests. In order to mitigate the burden of receiving many notification messages, DTL also supports a summary notification message for both directory requests and file requests. A summary notification includes the same information as the notification message described above except that it does not have the names of the files being moved. Such messages are delivered at a regular interval. Our experience indicates that the summary notification mechanism is more useful for scientists to move scientific dataset.

Determining the size of the thread pool is a challenging problem. Because the optimal value is affected by several factors and may change dynamically at runtime, automatic tuning is desired for optimal performance.

We use an adaptive scheme to tune the transfer thread pool size automatically. In the following text, we refer to a transfer source and destination as an "endpoint pair." We introduce a data structure, THREAD_POOL_MAP, that for each endpoint pair records the best-known number of transfer threads. When a new DR is initiated, DTL looks up THREAD_POOL_MAP. If an entry corresponding to the endpoint pair of this DR is found, the pool size is set to the recorded value; otherwise, it is set to an initial size (the default is eight).

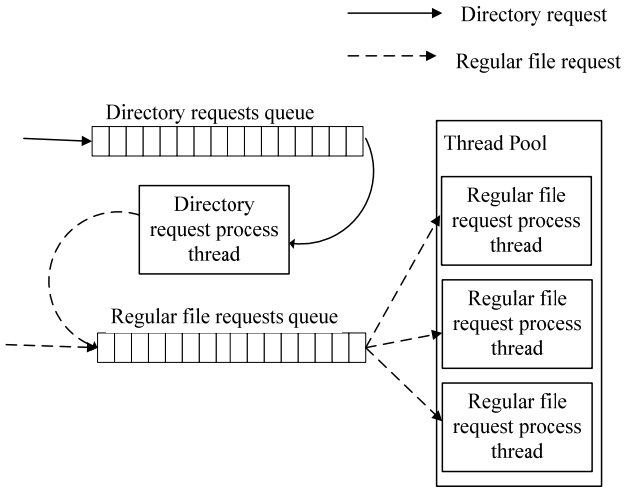


**Figure 2: Thread pool and queues in DTL**

The automatic tuning process periodically calculates the instantaneous throughput for each directory request. An average throughput is derived from five instantaneous throughput values. The thread pool expands (by default, adding four threads) if the current average throughput is larger than the preceding average throughput by some factor (the default is 1.3). If the current average throughput is smaller than the previous average throughput by some factor (default is 0.7), two situations are considered. If the current number of threads is larger than the previous number of threads, we regard the throughput deterioration as caused by congestion due to too many transfer threads, and we shrink the thread pool; redundant threads are killed after they finish their work. Otherwise, the throughput decrease is attributed to lack of transfer threads; hence, new threads are spawned and put into the pool. This process runs at a fixed interval to tune the thread pool size dynamically during runtime. When the directory transfer request completes, THREAD_POOL_MAP is updated with the current thread number. Our experiments show that this automatic tuning scheme can significantly improve data transfer throughput. Figure 3 describes this procedure in pseudocode.

The DTL program is designed to run on a client computer, which is more susceptible to unexpected errors such as machine reboot, power failure, or accidental shutdown of the program by a user. In addition, data transfers initiated by DTL may fail for various reasons, including disk failure and network outage. If a failure occurs while transferring a directory with a large number of files, it is not feasible to identify and retransfer the missing files manually. Thus, we implement in DTL a basic fault-tolerance mechanism that can handle client failures, server failures, and network failures.

For the errors that DTL can discover, such as a server crash or network outage, DTL retries several times at a user-specified interval. If all attempts fail, DTL writes the request to an error log file (error.log).

```
while (true)
  if(a new DR starts to be served)
    get endpoint pair from transfer request
    if (endpoint pair in THREAD_POOL_MAP)
      pool_size=get from THREAD_POOL_MAP
    else
      pool_size=default_pool_size
    end if
    thread pool size = pool_size
    prev_Throughput = 0
    prev_Threads=pool_size
    current_Threads=pool_size
  else
    for (i=1; i<=sampling_times;i=i+1)
      B1=bytes has been transferred at instant t1
      sleep for default_interval time
      B2=bytes has been transferred at instant t2
      ins_Throughputi=(B2-B1)/(t2-t1)
    end for
    AVG_Throughput=∑ins_Throughputi/sampling_times
    if(AVG_Throughput>expand_factor*prev_Throughput)
      prev_Throughput=AVG_Throughput
      prev_Threads=current_Threads
      expand thread pool size for the endpoint pair
    else
      if(AVG_Throughput<shrink_factor*prev_Throughput)
        if(prev_Threads>current_Threads)
          prev_Throughput=AVG_Throughput
          prev_Threads=current_Threads
          expand thread pool size for endpoint pair
        else
          shrink thread pool size for endpoint pair
        end if
      end if
    end if
    if (end of the DR reached)
     update THREAD_POOL_MAP with current_Threads
    end if
  end if
  sleep for a while
 end while
```

**Figure 3: Tuning procedure for thread pool size**

In contrast, DTL typically cannot detect or respond to client (DTL) failures. To permit recovery from such situations, we use a lightweight checkpoint-based error recovery mechanism. For each DR (including all nested subdirectories), four files are created for error recovery:

filecounts.log: records the number of files in the DR and includes a pointer (referred as "last file transferred pointer" in the following text) to the file transfer request that has the largest ID in all requests currently being processed;

filenames.log: records the source and destination of each file transfer request;

dircounts.log: records the total number of directories in the DR and how many have been processed;

dirnames.log: records the source and destination of each directory in the DR.

When DTL receives a DR, it writes the source and destination into dirnames.log and increases the total number of directories in dircounts.log by one. When subdirectories are retrieved and the corresponding DRs are constructed, dircounts.log and dirnames.log are updated in the same way. Filenames.log and the total number of files in filecounts.log are updated when a directory request is processed, and corresponding file transfer requests are constructed for files in the directory. After each directory transfer is completed, the processed directory number in dircounts.log is increased by one. The transfer thread updates the "last file transferred" pointer in filecounts.log right after it gets a file transfer request from FQ, and a checkpoint file is created for each file request at the same time. The name of the checkpoint file is the unique identifier (ID) of the file transfer request, There is no content in the checkpoint file; it is used only to record which files are being moved currently. When a transfer completes, the transfer thread deletes the checkpoint file.

Error recovery happens after DTL completes initialization. The error recovery procedure comprises four steps. First, a file transfer request is constructed for each error.log entry; second, a file transfer request is built for each check point file; third, the "last file transferred" pointer is obtained from filecounts.log, and a file transfer request is constructed for each filenames.log entry from the pointer until the end of the file; fourth, DTL gets DRs from dircounts.log and dirnames.log similarly. Figure 4 presents the pseudocode of the error recovery procedure.

```
for each entry in error.log
  construct a file transfer request
  put the file transfer request into FQ
end for
for each check point file
  get transferID of the check point file
  get corresponding entry from filenames.log
  construct file transfer request
  put the file transfer request into FQ
end for
p_value = the pointer value from filecounts.log
t_value=total number of files from filecounts.log
if(p_value < t_value)
  for each transferID in (p_value, t_value]
    get corresponding entry from filenames.log
    construct file transfer request
    put the file transfer request into FQ
  end for
end if
f_num = number of completed directories
t_num = total number of directories to transfer
if(f _num<t_num)
  for each dirID in (f_num, t_num]
    get corresponding entry from dirnames.log
    construct directory transfer request
    put the directory transfer request into DQ
  end for
  end if
```

**Figure 4: Error recovery procedure**

## 3.2 Data Scheduler

The data scheduler is responsible for ordering transfer requests according to a given scheduling policy and for putting requests into the DTL directory queue for actual data transfer. We designed three data scheduling policies to cater to the requirements of various scientific experiments.

The simplest policy, FCFS, adds file requests to the end of the file queue. In the case of a directory request, the data scheduler adds it to the end of the directory queue and recursively communicates with the GridFTP server to identify all nested subdirectories. Then, for each subdirectory, a directory request is constructed and appended to the directory queue. DTL is

responsible for expanding files under each subdirectory into the file queue and moving them.

Data generated by scientific experiments may have dependency relationships, and users might want to move one dataset before another. To this end, we designed a dependency-aware scheduling policy, DAS. In this policy, a user-specified dependency relationship is passed to the data scheduler with a DR. DAS behaves in the same way as FCFS except that the subdirectories returned by the GridFTP server are put into the directory queue according to the order specified by user in the dependency relationship. If only a subset of the subdirectories is named in the dependency relationship specification, the remaining subdirectories are put into the directory queue in the order they are returned from the GridFTP server.

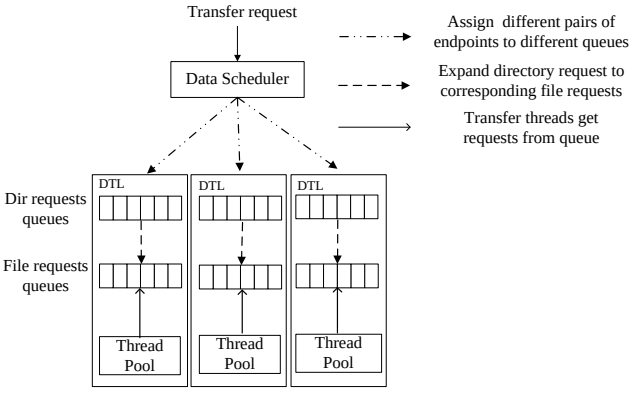


**Figure 5: Multiple-Pair Transfer Scheduling**

The third policy addresses scenarios in which data must be moved to different remote locations over different network links. Processing these transfer requests concurrently makes full use of the network links and can improve aggregate performance significantly. Thus, we designed the Multiple-Pair Transfer Scheduling (MPTS) policy, which, as illustrated in Figure 5, creates a DTL instance for each endpoint pair; hence, multiple endpoint pairs are served concurrently. In order to avoid exhausting the resources of the machine where the data movement system runs, at most only five concurrent DTL instances are allowed. If there are more than five endpoint pairs, those are appended to these DTL instances and processed sequentially. MPTS can be combined with DAS, in which case each endpoint pair has its own dependency relationship.

## 3.3 GridFTP GUI

The fourth component simplifies DTL usage by providing a graphical user interface. GridFTP GUI is a cross-platform GridFTP client tool based on Java web start technology [18] and can be accessed in a single click. Users can always get the most recent version of the application without any manual installation. Figure 6 is a screen snapshot of GridFTP GUI.

GridFTP GUI allows users to transfer files using drag-and-drop operations. Many scientific datasets are organized into a hierarchical directory structure. The total number of files under the top directory is typically large. However, the number of files in each nested subdirectory is moderate. The data scheduler and DTL handle the task of efficient data movement. GridFTP GUI is responsible for displaying the transfer status clearly and methodically. Users require well-organized information so that they can easily view and track the status of a transfer. Accordingly, we show transfer information for directories. For the directory that is being actively transferred, we list all files under that directory and show the status of each file. When all the files in the directory are transferred, the directory's status is updated to "Finished," and the files under that directory are removed from the display.

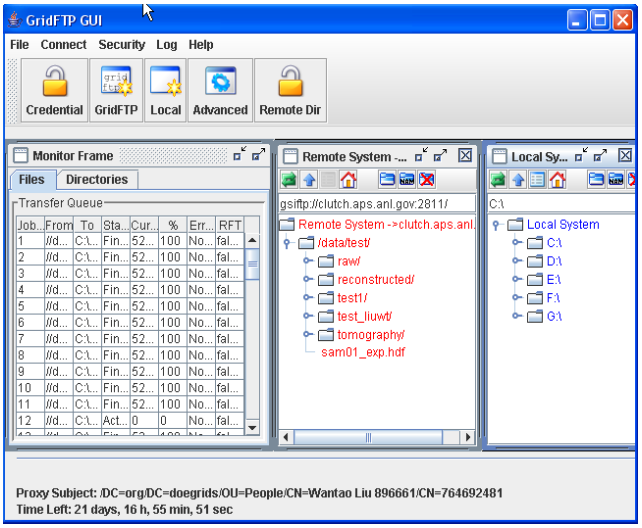


**Figure 6: GridFTP GUI screen snapshot**

Data produced by large-scale scientific experiments commonly is transferred among different organizations or countries, each having individual policies and trust certificates issued by distinct accredited authorities. Hence, establishing trust relationships is a big challenge.

The International Grid Trust Federation (IGTF) [20] is an organization that federates policy management authorities all over the world, with the goal of enhancing establishment of cross-domain trust relationships between Grid participants. The distribution provided by IGTF contains root certificates, certificate revocation list locations, contact information, and signing of policies. Users can download this distribution and install it to conduct cross-domain communication.

When GridFTP GUI starts up, it contacts the IGTF website and, if there is any update, downloads the latest distribution and installs it in the trusted certificates directory. This procedure ensures that the GUI trusts the certificates issued by the certificate authorities that are part of IGTF. This feature simplifies the establishment of cross-domain trust relationships with other Grid entities.

For more detailed information regarding GridFTP GUI, please refer to [25].

# 4. CASE STUDY

The Advanced Photon Source (APS) [3] at Argonne National Laboratory provides the Western Hemisphere's most brilliant x-ray beams for research. More than 5,000 scientists worldwide perform scientific experiments at the APS annually. Such experiments span all scientific disciplines, for example, improving vaccines against rotavirus, increasing operational efficiencies of aircraft turbine blades, and characterizing newly discovered superconducting materials. The efficient dissemination

of data acquired at the APS to remote scientific collaborators is of great concern. In this section, we describe how the tomography beamline at APS is making use of DTL.

```
Listener l = new DefaultListener();
DataTransferExecutor executor
   = new GridFTPTransferExecutor(
            Constants.DEFAULT_THREADS_NUM,
            l, "logfile");
executor.setSchedulePolicy(Constants.FCFS);
Transfer t1 = new DirTransfer(
           "gsiftp://clutch.aps.anl.gov:2811/data/tomo/",
           "gsiftp://qb1.loni.org:51000/work/tomo/ ");
 executor.addTransfer(t1);
```

**Figure 7: Tomo Script code snippet**

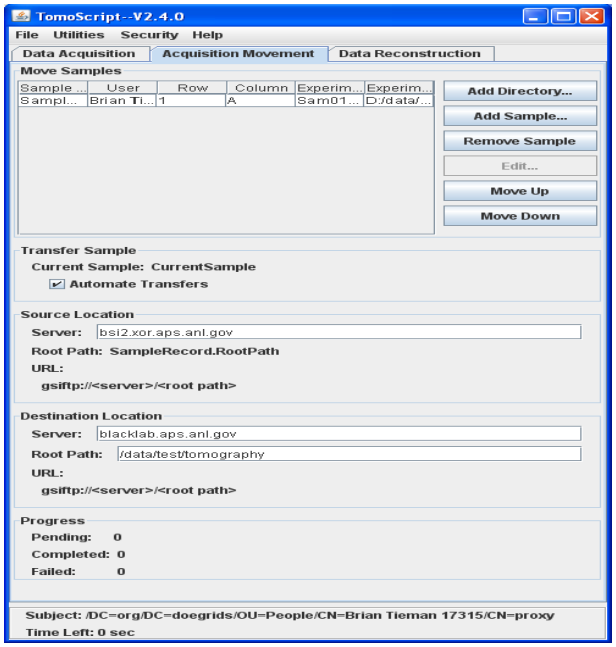


**Figure 8: Tomo Script code snippet, screen snapshot**

Developers at the APS have integrated the data scheduler and DTL into the Tomo Script program used to automate tomography experiments at the APS. Figures 7 and 8 show a code snippet and screen snapshot of Tomo Script. From the code snippet, we can see that the simple API implemented by our data transfer framework made this integration straightforward. Application developers create a DefaultListener instance for receiving notification messanges and a DataTransferExecutor instance (the implementation class of DTL) for data movement, then specify the data scheduling policy they want. Finally, they create a Transfer object with source and destination address and put it into the transfer queue. The instance of DataTransferExecutor will conduct the actual data transfer.

Tomo Script can acquire data while unattended for a group of samples loaded into an automated sample changer. It also can load samples into the x-ray beam and control all the equipment necessary to acquire the approximately 12 GB of data acquired per sample. In one 24-hour period, the system is capable of running 96 samples and acquiring 1.1 TB of data. This data must be moved to an on-site computational cluster for processing before scientists can determine whether critical acquisition parameters are correct for acquiring high-quality data. Tomo Script thus relieves the scientist of the arduous task of finding the right dataset out of hundreds to move and monitoring the data movement for proper completion. Transfer failures are noted, and automatic recovery is attempted. If multiple transfer failures occur, the scientist is alerted so appropriate action can be taken.

The other beamline users at APS are evaluating the framework, including GridFTP GUI, to simplify their data transfer work.

# 5. EXPERIMENTAL RESULTS

In this section we first describe the experiment configuration and then present our results.

## 5.1 Experiment Setup

We measured the time taken to transfer data between computers at the APS and both Louisiana State University (LSU) and the Pittsburgh Supercomputing Center (PSC). The GridFTP server machine at the three sites had the following configuration. The APS node is equipped with four AMD 2.4 GHz dual-core CPUs, 8 GB memory, and a gigabit Ethernet (1000 Mb/s) interface. The LSU node has two 2.33 GHz quad-core Xeon processors, 8 GB memory, and a gigabit Ethernet interface. The PSC node is with two 1.66 GHz dual-core processors, 8 GB memory, and a gigabit Ethernet interface. All these machines were running Linux with TCP autotuning enabled and configured with at least a 4 MB maximum TCP buffer. The network link between APS and LSU and the link between APS and PSC both traverse the public Internet. The round-trip time between APS and PSC is 32 ms, and the round trip time between APS and LSU is 33 ms.

**Table 1. Data size and file counts in the experimental data**

| Subdirectory Name | Subdirectory Size (GB) | Number of Files |
|---|---|---|
| tray01 | 140 | 20,549 |
| tray02 | 103 | 22,376 |
| tray03 | 34 | 4,335 |
| tray04 | 97 | 19,828 |
| tray05 | 19 | 2,432 |
| tray06 | 166 | 24,368 |
| jason_sam02 | 35 | 3,624 |
| tomo_1024 | 12 | 3,733 |
| tomo_2048 | 58 | 5,103 |
| tomo_2048_test | 35 | 2,918 |
| tomo_512 | 1.2 | 2,801 |
| Total | 696 | 112,093 |

We measure the performance of our data transfer library and data scheduler when run from the command line (DS_DTL) and when run from GridFTP GUI. For comparison, we also measure the performance of globus-url-copy (abbreviated here as GUC), a widely used GridFTP command-line client that does not incorporate the adaptive threading strategies implemented in DTL. Since the kernel on all the machines had autotuning enabled, manual configuration of the TCP buffer size was not necessary. GridFTP from Globus Toolkit 4.2.1 is installed at all three sites. All reported values are the mean of five trials.

Our experiments involved the movement of either a subset or all of a 696 GB dataset generated by a tomography experiment. This dataset comprises 97 samples and numerous files contained

in 11 subdirectories, as summarized in Table 1. Note that the average file size is small (only 6.2 MB), a common situation when dealing with experimental data.

## 5.2 Experimental Results

We present in Figure 9 data transfer times for DS_DTL and GridFTP GUI (both using the FCFS scheduling policy) and for GUC, when moving the tray04 directory (97 GB, 19,828 files) from APS to LSU. The performance of GUC improves with increasing concurrency value up to 20 concurrent threads; beyond that, the performance starts to degrade. This situation implies that using a flat, high-concurrent value throughout the transfer need not necessarily result in better performance. In contrast, DS_DTL's thread pool tuning procedure allows it to adjust dynamically the numbers of transfer threads used. As a result, it performs better than GUC. Moreover, when the concurrency value was increased beyond 16, intermittent failures occurred because of server load. GUC does not have robust failure-handling mechanisms to recover from those failures automatically. Indeed, if we consider only a stable GUC configuration (c<16), DS_DTL and GridFTP GUI perform much better. DS_DTL achieves an end-to-end transfer rate of roughly 277 Mbit/s. Because of the overhead of GUI elements and some synchronization operations, GridFTP GUI performs moderately worse than DS_DTL and the fastest GUC configuration. The DS_DTL thread pool size varied from 8 to 26 during the transfer, without much variation from run to run.

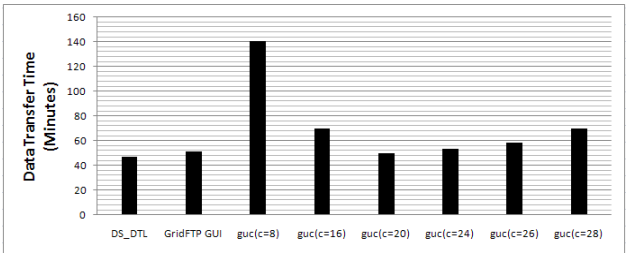


**Figure 9: Time taken by DS_DTL, GridFTP GUI, and GUC to move 97 GB in 19,828 files from APS to LSU. The numbers in parentheses represent the number of concurrent transfer processes used for different GUC configurations.**

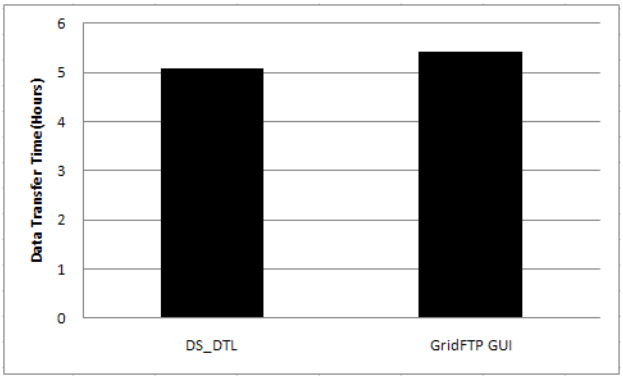


**Figure 10: Data transfer time for DTL and GridFTP GUI when moving 696 GB in 112,093 files from APS to LSU**

Figure 10 presents results for DS_DTL and GridFTP GUI when moving the full dataset from APS to LSU. (GUC did not run to completion for these large datasets because of transient system failures such as file system errors and server load.) Again, we see that GridFTP GUI incurs a modest overhead relative to raw DS_DTL.

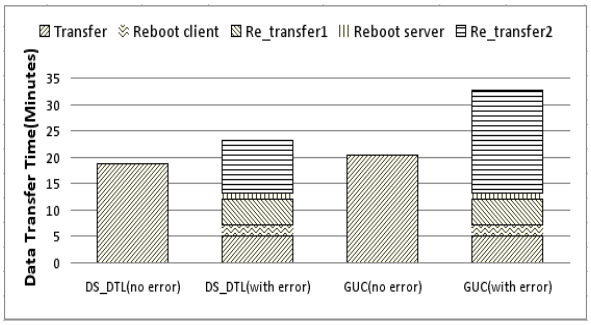


**Figure 11: Data transfer time of DS_DTL and GUC with client and server errors**

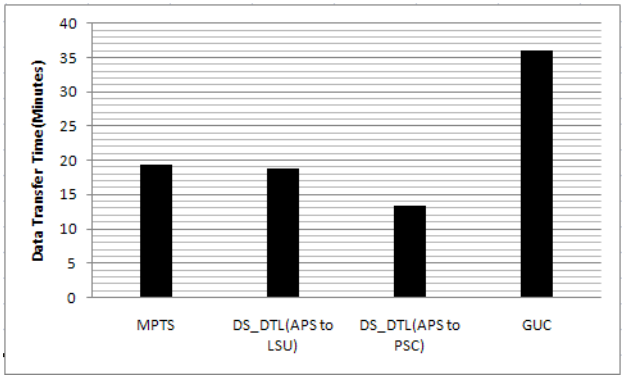


**Figure 12: Transfer times for DS_DTL with MPTS, sequential DS_DTL transfers, and GUC**

Our second experiment tested DS_DTL's error recovery capabilities. In this experiment, subdirectory tomo_2048 was moved from APS to LSU, with the FCFS scheduling policy. The retry interval was set to 30 seconds for DS_DTL, and the number of retries was set to five. The same parameters were set for GUC as well. We measured the response of both DS_DTL and GUC in the following scenario. Five minutes after transfer start, we rebooted the client computer, resulting in a 2-minute reboot process. Five minutes after the client computer reboot completed, we shut down the GridFTP server at LSU, restarting it one minute later.

The results are shown in Figure 11. DS_DTL handled the errors well, restarting correctly after the client computer reboot and resuming the interrupted transfer due to the GridFTP server shutdown when the server restarted. We see that the total time for DS_DTL with errors is slightly longer than that of DS_DTL without error plus the three minutes consumed by restarts. We attribute this discrepancy to the facts that (a) after program restart, the thread pool is not immediately optimal and (b) files that were in transit when the error happened must be retransferred.

In contrast, GUC restarted from scratch after the client machine reboot, wasting all effort performed prior to the reboot. GUC also did not handle the server shutdown well: the retry option did not have any effect, and GUC terminated immediately at server shutdown. Thus the time taken by the last restart (Re_transfer2 in the graph) of GUC with errors equals the transfer time without any error. In other words, all previous effort was wasted.

The last experiment evaluated the performance of the MPTS data scheduling policy. Two subdirectories were involved in this experiment: tomo_2048 was moved from APS to LSU, and jason_sam02 was moved from APS to PSC. We studied three scenarios: (a) DS_DTL when using MPTS to enable concurrent execution of the two subdirectory transfers; (b) sequential execution (FCFS scheduling policy is used here), again using DS_DTL, of first the LSU and then the PSU transfer; and (c) transfer using GUC. (GUC allows a user to request multiple transfers in one command, through the –f option, but the actual data movement is sequential.)

The results are shown in Figure 12. The times taken to move tomo_2048 from APS to LSU and jason_sam02 from APS to PSC individually are shown as the two columns in the middle of the graph. We see that the number corresponding to MPTS is only slightly greater than the maximum of those two times (the transfer time from APS to LSU). The time needed for GUC is the sum of the two individual transfers using GUC. MPTS significantly improves transfer performance.

## 6. CONCLUSION AND FUTURE WORK

We have presented a data transfer framework designed to meet the data transfer requirements of scientific facilities, which often face the need to move large numbers of relatively small files reliably and rapidly to remote locations. Building on GridFTP, this system uses a combination of automatic concurrency adaptation and restart mechanisms to move large volumes of data with high performance and robustness. Alternative scheduling policies support the specification of dependencies between transfers and the use of multiple network paths. The system has been deployed successfully in the Advanced Photon Source at Argonne National Laboratory for the transfer of experimental data.

Currently, GridFTP GUI cannot estimate the total and remaining transfer time of a request. We intend to add data transfer time estimation in the next release. We also plan to encapsulate this data transfer frramework in a Grid service with a standard interface, so that users can invoke these services from remote locations and conduct data transfers easily, without being aware of any updates to the service implementation or the data transfer framework.

## 7. ACKNOWLEDGEMENTS

This work was supported in part by the U.S. Department of Energy, under Contract DE-AC02-06CH11357.

## 8. REFERENCES

[1] CERN: http://www.cern.ch/.

[2] LIGO: http://www.ligo.caltech.edu/

[3] APS: http://www.aps.anl.gov/

[4] SNS: http://neutrons.ornl.gov/

[5] W. Allcock, “GridFTP: Protocol Extension to FTP for the Grid,” Global Grid Forum GFD-R-P.020, 2003.

[6] 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

[7] J. Postel and J. Reynolds, “File Transfer Protocol,” IETF, RFC 959, 1985

[8] PhEDEx: http://cmsweb.cern.ch/phedex/

[9] J. Rehn, T. Barrass, D. Bonacorsi, J. Hernandez, I. Semoniouk, L. Tuura, and Y. Wu, “PhEDEx High-Throughput Data Transfer Management System,” CHEP06, Mumbai, India, 2006

[10] R. O. Sinnott, C. Bayliss, C. Davenhall, B. Harbulot, M. Jones, C. Millar, G. Roy, S. Roy, G. Stewart, J. Watt, and A. Aseno, “Secure, Performance-Oriented Data Management for nanoCMOS Electronics,” eScience 2008

[11] caGrid: http://cagrid.org/display/cagridhome/Home

[12] caGrid Transfer: http://cagrid.org/display/transfer/Home

[13] caGrid Bulk Data Transfer: http://cagrid.org/display/bdt/Home

[14] T. Kosar and M. Livny, “Stork: Making Data Placement a First Class Citizen in the Grid,” 24th International Conference on Distributed Computing Systems (ICDCS 2004), Tokyo, Japan, March 2004.

[15] T. Kosar and M. Balman, “A New Paradigm: Data-Aware Scheduling in Grid Computing,” Future Generation Computer Systems, 25, no. 4, April 2009, pp. 406-413

[16] N. Ali and M. Lauria. “Improving the Performance of Remote I/O Using asynchronous Primitives,” 15th IEEE International Symposium on High Performance Distributed Computing, 2006, pp. 218–228

[17] CoG jglobus:http://dev.globus.org/wiki/CoG_jglobus

[18] Java web start technology: http://java.sun.com/javase/technologies/desktop/javawebstart/index.jsp

[19] RFT:http://globus.org/toolkit/docs/latest-stable/data/rft/#rft

[20] IGTF, http://www.igtf.net/

[21] S. Vazhkudai, "Bulk Data Transfer Forecasts and Implications to Grid Scheduling,” Ph.D. Dissertation, The University of Mississippi, May 2003

[22] T. J. Hacker, B. D. Noble, and B. D. Atley. “The end-to-end performance effects of parallel TCP Sockets on a Lossy Wide Area Network,” IPDPS ’02, p. 314. IEEE, April 2002.

[23] D. Lu, Y. Qiao, P. A. Dinda, and F. E. Bustamante. “Modeling and Taming Parallel TCP on the Wide Area Network,” IPDPS ’05, p. 68.2. IEEE, April 2005.

[24] Esma Yildirim, Mehmet Balman, and Tevfik Kosar, “Dynamically Tuning Level of Parallelism in Wide Area Data Transfers,” International Workshop on Data-aware Distributed Computing, pp. 39–48, June 2008.

[25] Wantao Liu, Rajkumar Kettimuthu, Brian Tieman, Ravi Madduri, Bo Li and Ian Foster, “GridFTP GUI: An Easy and Efficient Way to Transfer Data in Grid,” GridNets2009, September 2009.

[26] Branco, M., Cameron, D., Gaidioz, B., Garonne, V., Koblitz, B., Lassnig, M., Rocha, R., Salgado, P. and Wenaus, T. “Managing ATLAS data on a petabyte-scale with DQ2,” Journal of Physics: Conference Series, Volume 119, Issue 6, pp. 062017 (2008).

[27] gLite File Transfer Service: www.gridpp.ac.uk/wiki/GLite_File_Transfer_Service