Moving huge scientific datasets over the Internet

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/Argonne National Laboratory, 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 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 dataset 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 dataset 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 for handling 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.

KEY WORDS: data transfer; data intensive computing; Internet; e-science

1. INTRODUCTION

Scientists from diverse disciplines are facing a data deluge. They are running scientific experiments, for example, at CERN [1], LIGO [2], the Advanced Photon Source (APS) [3], and the Spallation Neutron Source [4] that generate multiple gigabytes to terabytes of data every day. Data-intensive science recently has been called the fourth paradigm in scientific research; the first three are theory, experiment, and computer simulation [5] [6]. Frequently, scientific datasets must be disseminated over the Internet to remote collaborators for replication or to computational centers capable of running the complex, CPU-intensive applications needed to analyze the data. For example, CERN organizes its data centers as a three-tiered structure distributed around the globe [7]. One tier-0 center performs the initial processing of all experimental data and distributes this data to eleven tier-1 centers. These tier-1 centers are located in different countries and continents; they are equipped with sufficient computing power and data storage for data processing. After this, the data that is of interest to individual scientists is moved to a tier-2 center for specific analysis tasks. The APS at Argonne National Laboratory [3] provides another example for large volume data transfer over wide-area network. More than 5,000 scientists worldwide perform scientific experiments at the APS annually. However, APS is not a data center, and it does not provide adequate storage capacity for keeping all the data a long time. Hence, the experiment data has to be moved to other places quickly.

Transferring large volumes of data on physical media such as tapes or removable disk drives is an approach named Sneakernet [8]. It can be used in situations where there is no good network connection for the data transfer or where the data is sensitive. Seti@home uses this approach to move data from the Arecibo Observatory in Puerto Rico to Berkeley [9]. However, this approach 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 worldwide. 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. Various protocols and tools such as scp and FTP [10] have been developed for transferring data over the Internet. GridFTP [11][12] is widely used for transferring bulk data over wide-area networks. It extends standard FTP for high-performance operation and security For example, the high energy physics community bases its entire tiered data movement infrastructure for the Large Hadron Collider computational 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.

Currently, scientists face a number of challenges with data movement:

- Performance: typically, scientists use the default parameter values in their commands or scripts. Tuning parameters optimally is not an easy job that can be daunting to users. Moreover, since these parameter values are affected by various runtime factors, dynamic adjustment is desirable.
- Fault tolerance: Data transfer over the Internet is error-prone. Errors and interruptions during data transfers are inevitable obstacles. It is not acceptable to always restart the transfer from the beginning.

- Transfer status monitoring: Scientists usually move a dataset comprising numerous directories and files once. However, basic commands or scripts are not capable of monitoring detailed transfer status, for example, which files are being moved currently and how much data remains.
- Easy-to-use interface: Both commands and scripts require some computer knowledge to study and use. Scientists require a tool with an easy-to-use interface, so that they can concentrate more on their research work.

Based on this analysis, we conclude that scientists desire a high-performance, straightforward, user-friendly and robust data transfer mechanism that can significantly improve their work efficiency. Motivated by these considerations, we have designed and implemented a data transfer framework.

This paper makes four contributions: (1) a data transfer framework architecture that addresses the requirements just listed; (2) an algorithm to autotune data transfer concurrency and can improve performance significantly; (3) four data scheduling algorithms; and (4) an error recovery algorithm that addresses both client-side, server-side and network 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 [13][14] data transfer management system is used by the CMS experiment at CERN. PhEDEx consists of a set of agents responsible for file replication, routing decisions, tape migrations, and so on. In PhEDEx, data transfer and placement decisions are made in terms of datasets, which are composed of hundreds to thousands of files; however, in our system, not only datasets but also individual files are supported. PhEDEx cannot recover from client crash, whereas our system can.

The caGrid [15] 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 [16] is used for moving small amounts of data between a client and server, and BulkData Transfer [17], based on GridFTP, is used for moving huge amounts of data. 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 deal with client, network, and server errors.

Sinnott et al. [18] 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 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 [19][20] 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. Our system implements some different scheduling algorithms, for example, a multipair transfer scheduling policy and round-robin scheduling policy. Stork supports multiple data transfer protocols and can decide which protocol to use at runtime. It also implements a basic error recovery mechanism through retry and kill-and-restart mechanisms. However, it cannot recover from client crash, whereas our system can.

Ali and Lauria [21] 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) [22] 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.

The gLite File Transfer Service [23] 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 its own data scheduling policies. The gLite FTS system uses an Oracle database to hold the transfer state, while our system put it into a plain file, which is more lightweight.

FOBS [24] is a user-level communication protocol proposed for large-scale data transfer in high-bandwidth, high-delay network environment. It uses UDP as underlying transport layer protocol, and leverages acknowledgments and retransmission for reliability. FOBS can utilize available network bandwidth efficiently; hence data transfer performance is improved.

Vazhkudai [25] 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. [26] 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. [27] and Yildirim et al. [28] 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

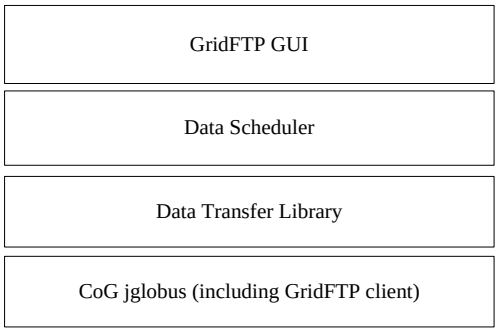

Figure 1: Architecture of the data transfer framework

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 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 the 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 [29] library includes a pure Java GridFTP client API; it can be used to build applications that communicate with GridFTP servers.

In the subsections 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.

3.1.1 Protocol Adaptor

Different scientific experiments or facilities use distinct data transfer protocols. It is not feasible to require all scientists to use the same data transfer protocol. Hence, we designed a protocol adaptor layer to incorporate various data transfer protocols.

Users who plan to add support to a new protocol simply need to implement two Java interfaces:

TransferTask: the representation of an executable transfer task. It consists of information required to conduct a transfer. Each kind of transfer mechanism should provide an implementation of this interface.

DataTransferExecutor: The concrete implementation of this interface conducts the actual data transfer using a specific data transfer protocol. TransferTasks waiting in the queue are passed to it. Each transfer thread has its own instance of DataTransferExecutor.

Because scientists typically transfer one or more datasets containing numerous files, in our current implementation based on GridFTP we reuse network connections to improve performance and reduce system resource overhead. An established GridFTP connection is kept in the DataTransferExecutor and used for data transfer until it does not match the source and destination of the TransferTask.

3.1.2 Asynchronous Data Transfer

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 types of queues: file queue (FQ) for file transfer requests and directory queue (DQ) for directory transfer requests. DTL has only one DQ. 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. By default, only one FQ is initiated. However, specific scheduling policies (e.g., round-robin introduced in Section 3.2.4) require multiple FQs. In that case, files from different directory requests are put into distinct FQs. Each file transfer request in an FQ 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 the 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 the FQ. The file transfer request process thread in the pool repeatedly gets an FR from the 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 datasets.

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).

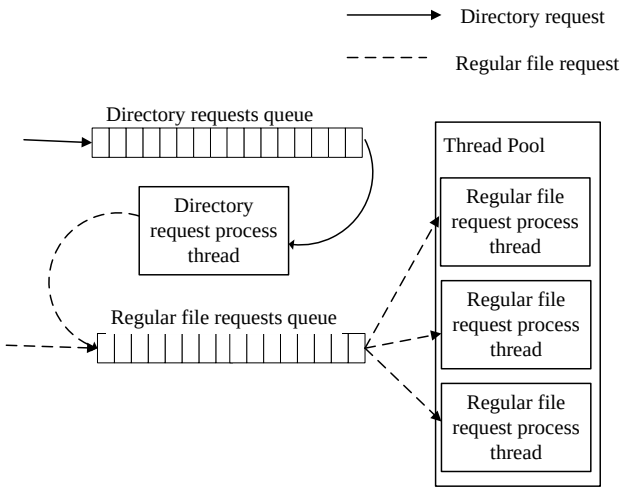

Figure 2: Thread pool and queues in DTL

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.

3.1.3 Fault Tolerance

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).

In contrast, DTL typically cannot detect or respond to client 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.

```
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

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; and fourth, DTL gets DRs from dircounts.log and dirnames.log similarly. Figure 4 presents the pseudocode of the 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. Different scheduling policies apply to different user scenarios. In this section, we present four data scheduling policies designed to meet the requirements of various scientific experiments.

3.2.1 First-Come, First-Served

The simplest policy, first-come, first-served (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.

```
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

3.2.2 Dynamic Priority

Data generated by scientific experiments may have priorities: some datasets are more important than others. The dataset with highest priority should be moved first. To this end, we designed a dynamic priority (DP) scheduling policy. There are 10 priority levels, from 0 to 9, where 0 represents the highest priority and 9 the lowest. The user specifies a priority number when submitting a transfer request, and DP finds an appropriate position for the request. The new request will preempt the transfer request that is being processed, if the new request priority is higher. All transfer requests in the queue are ordered according to their priorities.

One well-known drawback of fixed priority is “starvation”, which means transfer requests with low priority in the queue are always delayed and cannot get serviced for a long time. This situation deteriorates job turnaround time and user experience significantly. In order to overcome this issue, DP periodically checks requests waiting in the queue, then dynamically increases their priority based on their waiting time and changes its location in the queue accordingly.

3.2.3 Multiple-Pair Transfer

Large-scale science facilities typically accommodate collaborators around the world. These collaborators need to move scientific data back to their home institution for further analysis. In this case, data flows 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, the number of DTL instances allowed is restricted by a configuration parameter. If the number of endpoint pairs exceeds this restriction, those are appended to these DTL instances and processed sequentially.

3.2.4 Round-Robin Transfer

Fairness is an important metric in scheduling and for satisfactory user experience if multiple users put requests into the same directory queue. However, FCFS cannot guarantee it. The directory transfer request at the head of the queue occupies all network link capacity. If this request takes a lot of time, then the following short requests suffer from long waiting time and turnaround time.

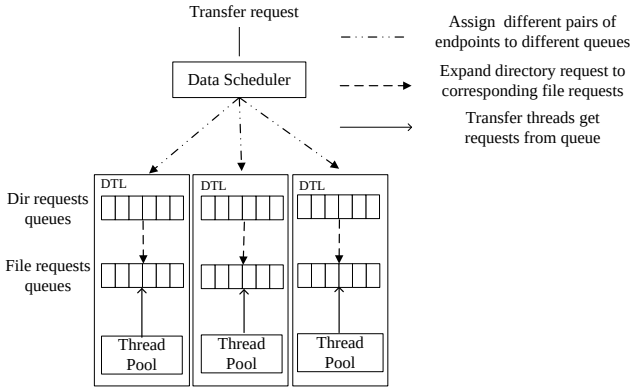

Figure 5: Multiple-Pair Transfer Scheduling

```
While (there are non-empty queues)
  For each FQ
    current_queue = current FQ
    if (current_queue is non-empty)
       quota = quota of the current_queue
       file_size = size of the file at head of current_queue
       if (quota < file_size)
           quota = quota+DAFAULT_QUOTA
           save quota for current FQ
       else
           quota = quota – file_size
           file = dequeue the head element of current_queue
           start transfer file
       endif
    endif
  end for
end while
```

Figure 6: Pseudo code of round-robin scheduling policy

The round-robin (RR) transfer scheduling policy tries to guarantee the fairness between multiple ongoing transfers. If the request at the head of the queue takes long time to transfer and following requests are short, the round-robin policy can improve the response time of the short transfer requests. In RR, the scheduler creates an FQ for each directory request. All files that belong to the directory request are put into the FQ. The transfer threads serve each FQ in a round-robin fashion. A configurable quota is used for restricting how much data the transfer threads can move from an FQ in a round (the initial quota value for all queues is the same). When the remaining quota of an FQ is

not big enough for moving the next file in the queue, the remaining quota is cumulated for the next round, and the transfer threads start to transfer files from the next FQ. Figure 6 demonstrates this process.

Both MPTS and RR are designed for the multiple-user scenario. The MPTS and RR policies differ in two ways:

1) MPTS consumes more client resources, since it creates multiple instances of DTL and there will be a lot of transfer threads.
2) If all the multiple users plan to move their data from the same source to the same destination, MPTS probably is not a good choice. It makes contention for network resource. In this case, round-robin is better.

3.3 GridFTP GUI

The fourth component of our new framework simplifies DTL usage by providing a graphical user interface. GridFTP GUI is a cross-platform GridFTP client tool based on Java web start technology [30] and can be accessed in a single click. Users can always get the most recent version of the application without any manual installation. Figure 7 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 typically is 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.

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) [31] 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 see [32].

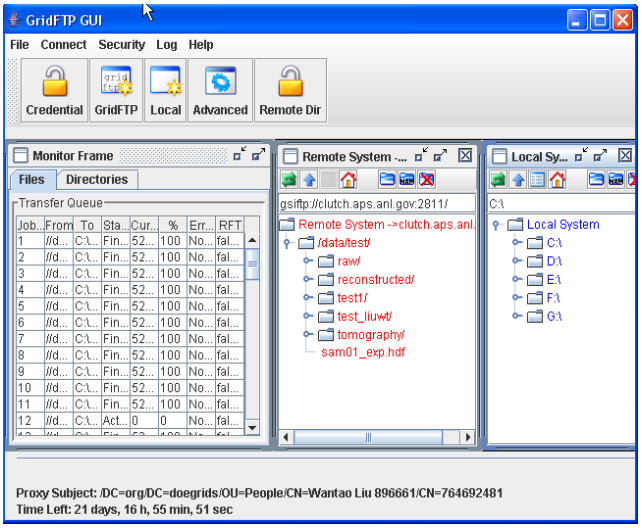

Figure 7: GridFTP GUI screen snapshot

4. CASE STUDY

The Advanced Photon Source [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.

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 8 and 9 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 messages and a DataTransferExecutor instance (the implementation class of DTL) for data movement and then specify the data scheduling policy they want. Next, 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.

```
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 8: Tomo Script code snippet

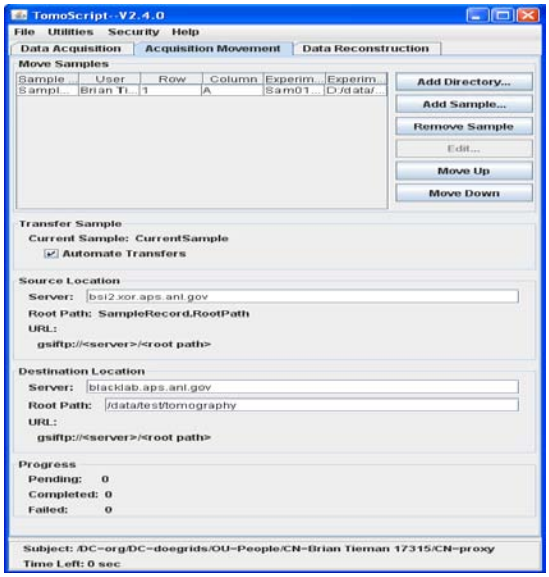

Figure 9: Tomo Script code snippet, screen snapshot

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.

We measure the performance of our data transfer library and data scheduler when run from the command line (DS_DTL) and 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.

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

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 10 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.

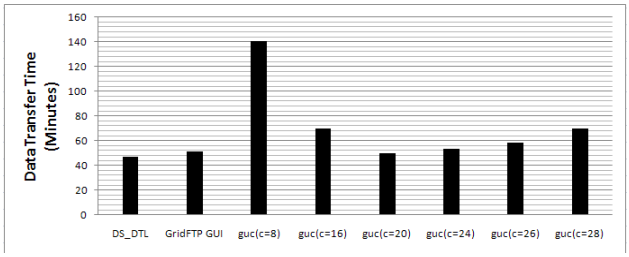

Figure 10: Time taken by DS_DTL, GridFTP GUI, and GUC to move 97 GB in 19,828 files from APS to LSU. The numbers in parentheses represent the number of concurrent transfer processes used for different GUC configurations.

Figure 11 depicts the changes of number of threads in DTL during this experiment. We can see that the number of threads is dynamically adjusted during runtime, fluctuating from 8 to 26. The number of threads increases to 16 very quickly (in about 1 minute); then the pace of change slows, and the number of threads stays above 16 until the transfer finishes. This figure shows that our threads-adjusting algorithm can reach high transfer performance quickly and maintain it during the entire transfer.

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. 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 two-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 12. DS_DTL handled the errors well, restarting correctly after the client computer reboot and resuming the transfer interrupted by 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.

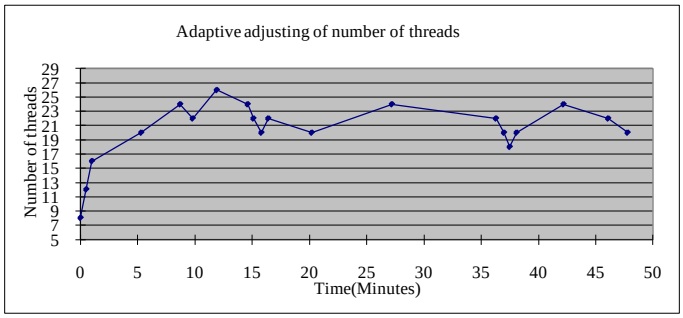

Figure 11: Changes of number of threads in DTL during the transfer of tray04

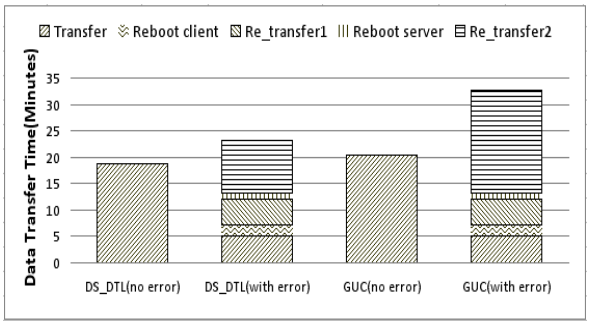

Figure 12: Data transfer time of DS_DTL and GUC with client and server errors

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 had no 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 third 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: (1) DS_DTL when using MPTS to enable concurrent execution of the two subdirectory transfers; (2) sequential execution (FCFS scheduling policy is used here), again using DS_DTL, of first the LSU and then the PSU transfer; and (3) 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 13. 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.

The fourth experiment examined the performance of dynamic priority and round-robin scheduling policies. Tray03 and tray05 were moved from APS to LSU. With the round-robin scheduling policy, the default quota for both datasets was set to 400 MB. With the dynamic priority scheduling policy, Tray03's priority was set to 9, the lowest priority; and Tray05's priority was set to 0, the highest priority. We first requested tray03 and then requested tray05 one minute later. Table 2 shows the turnaround time and wall clock time of the FCFS, round-robin, and dynamic priority scheduling policies. Turnaround time is computed as the difference between transfer finish time and the job submission time. Wall clock time is the time DTL takes to finish its requests; in this case, it is the time takes to finish tray03 and tray05.

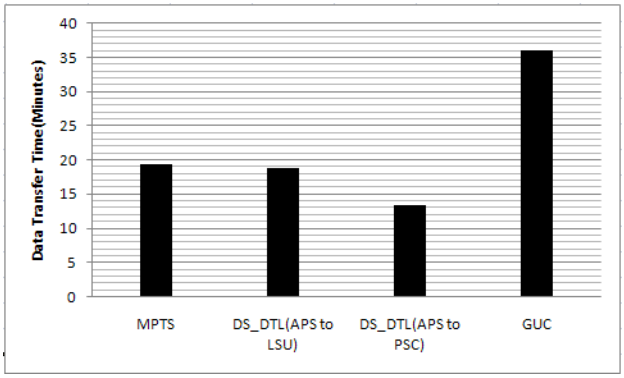

Figure 13: Transfer times for DS_DTL with MPTS, sequential DS_DTL transfers, and GUC

From Table 2, we can see that the wall clock time for the three scheduling policies is similar; but the finish order of transfer requests is different. For FCFS, DTL starts tray05 after finishing tray03; for dynamic priority scheduling policy, tray03 is being moved before tray05 is submitted. When tray05 is requested, since its priority is higher than tray03's priority, tray03 is preempted and tray05 is moved first. After trray05's completion, the remaining part of tray03 is transferred. With the round-robin scheduling policy, tray03 occupies the network resource exclusively before tray05 is put into the queue. Then, tray03 and tray05 are interleaved, and they share the network capacity. Since the size of tray05 is about half of that of tray03, tray05 finished earlier than tray03 in round-robin even though it was requested later. Because of the overhead of switching queues, the wall clock time of round-robin is a little larger than that of FCFS, but the distinction is not obvious. Figure 14 shows data volume transferred in each round. Since the directory request process thread takes some time to recursively retrieve file information under the directory and populates the FQ, transfer threads drains FQ at the very beginning of the transfer. Hence, the first several round moves little data. After that, the data volume moved in each round fluctuates around 400MB, which is set as the default quota.

Table 2. Turnaround time and wall clock time (seconds) comparison of FCFS and round-robin

	FCFS	Dynamic Priority	Round Robin
Tray03 (turnaround time)	683	1019	1043
Tray05 (turnaround time)	962	392	677
Total wall clock time	1022	1019	1043

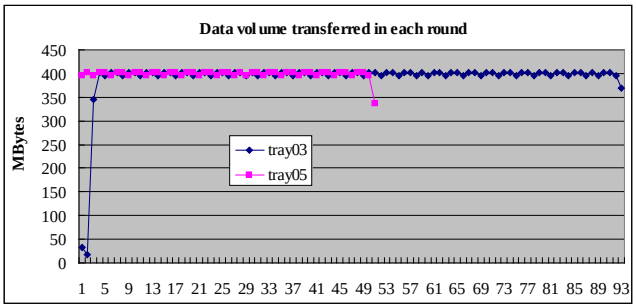

Figure 14: Data volume transferred in each round in round-robin scheduling policy, both tray03 and tray05 are moved from APS to LSU, and tray05 are requested one minutes later than tray03

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 framework 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.

ACKNOWLEDGMENTS

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

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] G. Bell, T. Hey, and A. Szalay, “Computer Science: Beyond the Data Deluge,” Science, 323, March 6, 2009, pp. 1297–1298

[6] T. Hey, S. Tansley, and K. Tolle, *The Fourth Paradigm: Data-Intensive Scientific Discovery*, Microsoft Research, 2009

[7] WLCG tier sites: http://lcg.web.cern.ch/LCG/public/tiers.htm

[8] J. Gray, W. Chong, T. Barclay, A. Szalay, and J. Vandenberg, “Terascale Sneakernet: Using Inexpensive Disks for Backup, Archiving, and Data Exchange,” Arxiv preprint cs/0208011, 2002

[9] E. Korpela, D. Werthimer, D. Anderson, J. Cobb, and M. Lebofsky, “Seti@ home—Massively Distributed Computing for Seti,” Computing in Science and Engineering, 3, no. 1, 2001, pp. 78–83

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

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

[12] 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

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

[14] 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

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

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

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

[18] 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

[19] 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

[20] 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

[21] 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

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

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

[24] P. Dickens and W. Gropp, “An Evaluation of Object-Based Data Transfers on High Performance Networks,” Proceedings of the 11th Conference on High Performance Distributed Computing, 2002

[25] S. Vazhkudai, "Bulk Data Transfer Forecasts and Implications to Grid Scheduling," Ph.D. dissertation, University of Mississippi, May 2003

[26] 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, IEEE, April 2002, p. 314

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

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

[29] CoGjglobus:http://dev.globus.org/wiki/CoG_jglobus

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

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

[32] 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