A Framework for Self-optimizing, Fault-tolerant, High Performance Bulk Data Transfers in a Heterogeneous Grid Environment

Tevfik Kosar, George Kola and Miron Livny
Computer Sciences Department, University of Wisconsin-Madison
1210 West Dayton Street, Madison WI 53706
{kosart,kola,miron}@cs.wisc.edu

Abstract

The drastic increase in the data requirements of scientific applications combined with an increasing trend towards collaborative research has resulted in the need to transfer large amounts of data among the participating sites. The general approach to transferring such large amounts of data has been to either dump data to tapes and mail them or employ scripts with an operator at each site to baby-sit the transfers to deal with failures. We introduce a framework which automates the whole process of data movement between different sites. The framework does not require any human intervention and it can recover automatically from various kinds of storage system, network, and software failures, guaranteeing completion of the transfers. The framework has sophisticated monitoring and tuning capability that increases the performance of the data transfers on the fly. The framework also generates on-the-fly visualization of the transfers making identification of problems and bottlenecks in the system simple.

1. Introduction

With the increase in collaborative research, the amount of data that has to be transferred among participating sites is increasing. In many cases, due to the lack of a common interface and the know-how to perform high performance bulk data transfers, researchers have resorted to dumping of data to tapes and shipping them via Federal Express [7].

Owing to the various data grid initiatives [12] [20], the underlying network capacity has increased enough to be able to support such bulk data transfers. Many of the large collaborative research initiatives like CMS [4] have multiple participating sites and need to move data among them. Further, the different storage sites employ different mass-storage systems which do not have a common interface. The general approach in this case has been to use scripts along with having an operator at each site to baby-sit the transfers.

We have designed and implemented a framework that fully automates data transfers between heterogeneous systems. The framework supports most of the protocols used for data transfers and makes it simple to support new protocols. The framework performs dynamic tuning of protocol parameters depending on the environment conditions improving the performance of data transfers on the fly. The framework is highly resilient and can automatically recover from a variety of network, storage system and software failures.

2. Related Work

Allcock et al [1] introduce the GridFTP protocol and Replica Catalog and discuss how they can be used for secure and efficient data transfer and data replication. The Reliable File Transfer Service(RFT) [19] allows byte streams to be transferred in a reliable manner. It is able to handle a wide variety of problems like dropped connections, machine reboots, and temporary network outages automatically via retrying. Kangaroo [21] provides high throughput wide-area data movement for remotely executing jobs by overlapping CPU and I/O. Kangaroo also has a certain degree of fault tolerance to cope with failures that occur in the wide-area. GridFTP, RFT and Kangaroo are tools that can be used to move data between two end-points supporting their interface. They cannot be used to move data between heterogeneous storage systems lacking a common interface. In addition, they do not have network monitoring and auto-tuning capabilities.

Feng [7] mentions a case where visualization scientists at Los Alamos National Laboratory dump data to tapes and send them to Sandia National Laboratory via Federal Express as it is faster than electronically transmitting them via TCP over the 155 Mbps(OC-3) WAN backbone.

The Lightweight Data Replicator (LDR) [14] can replicate data sets to the member sites of a Virtual Organization or Data Grid. It was developed for replicating LIGO [16] data, and in its present form, LDR expects the use of a single data transport protocol (GridFTP). Our framework is more general in nature. It allows data movement between systems not supporting a single data transport protocol. Our framework is flexible and can be used to perform a variety of data management operations including replication.

3. Framework

The framework brings together a set of tools for enabling data transfers in a heterogeneous environment. We have leveraged existing tools and have built new ones to complement them. We list the tools used in the framework and explain the functions they perform.

Condor. The Condor [17] workload management system was selected as the scheduler for computational jobs in our framework. Condor provides a job queuing mechanism and resource monitoring capabilities. It allows the users to specify scheduling policies and enforce priorities. Condor has an extension called Condor-G [9], which allows users to submit their jobs to inter-domain resources by using the Globus Toolkit [8] functionality. In this way, user jobs can get scheduled and run not only on Condor resources but also on PBS [10], LSF [22], LoadLeveler [11], and other grid resources.

Stork. We have used Stork [15] as the scheduler for data transfer jobs. Stork is a specialized scheduler for data placement activities in heterogeneous environments. Data placement encompasses all data movement related activities such as transfer, staging, replication, space allocation and deallocation. Stork can queue, schedule, monitor, and manage data placement jobs and ensure that the jobs complete.

Directed Acyclic Graph Manager(DAGMan). We use a Directed Acyclic Graph(DAG) to represent data movement in our framework. In a DAG, jobs are represented as nodes and the dependencies between jobs are represented as directed arcs between the respective nodes.

Figure 1 shows a sample data movement with computation. Data from a source S is moved to execute site E. E transforms the data and sends the result to destinations D1, D2 and D3. Figure 2 shows the DAG for the sample data movement with computation. There are nodes(jobs) to remove the data from the intermediate nodes after the data movement. We feel that the DAG Model is very flexible and gets its power from being able to run computation alongside data movement. It is possible to perform sophisticated processing using the DAG. For example the Destination D2 may want only a subset of data from E. In this case, D2 can upload a filter which will determine whether the file should be transferred to D2. Also it would be possible to run computation on the result generated at E to determine where the file should go to. This is very useful in real-life where data with certain characteristics is useful to certain sites and a computation needs to be run on the data to determine if it has those characteristics. This feature eliminates the need to transfer all the data and then discard the irrelevant ones.

To perform the management of the DAGs, we employed the Directed Acyclic Graph Manager (DAGMan) [5] which is a service for executing multiple jobs with dependencies between them. DAGMan accepts a declaration that specifies the jobs to be executed and the order of their execution. It logs the execution of the DAG to persistent storage, allowing it to resume a DAG where it left off, even in the face of crashes and other failures.

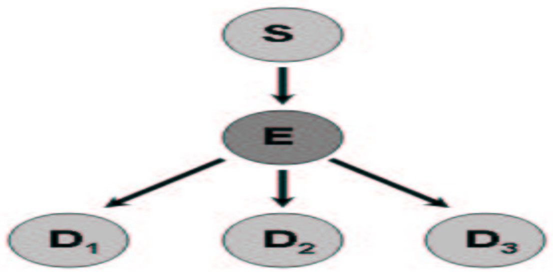

Figure 1. A Sample Data Movement with Computation. ***Data from source S is transferred to execute site E. E performs a computation on the data and sends the result to destinations D1, D2 and D3.***

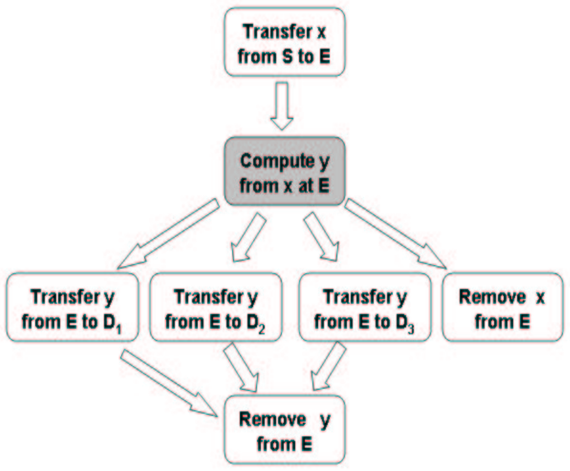
Transfer x from S to E
Compute y from x at E
Transfer y from E to D_1
Transfer y from E to D_2
Transfer y from E to D_3
Remove x from E
Remove y from E

Figure 2. DAG for the Sample Data Movement with Computation. ***Data X is transferred from source S to execute site E. E performs a computation on the data X generating data Y. Data Y is sent destinations D1, D2 and D3.***

We have introduced the concept of data placement jobs to DAGMan. It can differentiate between computational jobs and data placement jobs, and then submit computational jobs to Condor/Condor-G and the data placement jobs to Stork. The progress of both computational and data placement jobs can be monitored through user log files of Condor and Stork.

Environment Monitoring Infrastructure. We have developed an environment monitoring infrastructure that monitors the different environmental parameters affecting data transfers. In its present form, the infrastructure monitors disk, memory and network characteristics. We have developed memory and I/O profilers and built a network profiler using existing tools.

The memory profiler determines the optimal memory block size to be used to copy data and the memory copy bandwidth. The I/O profiler determines the optimal read and write block sizes and the increment block size that can be added to the optimal value to get the same performance and the disk bandwidth at the optimal value. The I/O profiler takes a list of pathnames as input, so that it can determine the characteristics of different disks in a multi-disk system.

The network parameters measured are end-to-end bandwidth, end-to-end latency, number of hops, the latency of each hop. The kernel TCP parameters are also determined from /proc on linux. Since end-to-end measurement requires two hosts, this measurement is done between every pair of hosts that may transfer data between each other. We used traceroute to get the number of hops and the latency between hops.

For end-to-end bandwidth measurement, we used a combination of intrusive and non-intrusive techniques. For the non-intrusive measurement, we used pathrate which uses packet dispersion techniques to estimate the bottleneck bandwidth. The intrusive method uses actual data transfers. First, we used packet dispersion technique to estimate the bandwidth and then we performed actual transfers using DiskRouter transfer tool to get the available bandwidth. If the numbers are widely different, we perform a series of experiments to extract the correlation between the two. Once this initial setup is done, we always use the non-intrusive technique. While the initial measurement is long, subsequent periodic measurements are light-weight and do not perturb the system.

The different profilers run as condor jobs on the respective nodes.

Parameter Tuner. The Parameter Tuner uses the information collected by monitoring infrastructure and tries to determine the optimal I/O block size, TCP buffer size and the number of TCP streams for the data transfer from a given node X to a given node Y.

It calculates the TCP buffer size as the bandwidth delay product. For the number of parallel streams, it uses a heuristic :it adds an extra stream for each hop with a latency greater than 10ms. If there are multiple streams and the number of streams is odd, it rounds it off to an even number. The reason for doing this is that some protocols do not work well with an odd number of streams. For each stream, it sets the TCP buffer size as the quotient of the bandwidth delay product divided by the number of streams. This ensures that this scheme does not cause congestion in steady state.

The reason for adding a stream for every 10ms hop is as follows: In a high-latency multi-hop network path, each of the hops may experience congestion independently. If a bulk data transfer using a single TCP stream occurs over such a high-latency multi-hop path, each congestion event would shrink the TCP window size by half. Since this is a high-latency path, it would take a long time for the window to grow, with the net result being that a single TCP stream would be unable to utilize the full available bandwidth. Having multiple streams reduces the bandwidth reduction of a single congestion event. Most probably only a single stream would be affected by the congestion event and halving the window size of that stream alone would be sufficient to eliminate congestion. The probability of independent congestion events occurring increases with the number of hops. Since only the high-latency hops have a significant impact because of the time taken to increase the window size, we added a stream for all high-latency hops and empirically found that hops with latency greater than 10 ms fell into the high-latency category.

The Parameter Tuner understands kernel TCP limitations. Some machines may have a maximum TCP buffer size limit less than the optimal needed for the transfer. In such a case, the parameter tuner uses more streams so that their aggregate buffer size is equal to that of the optimal TCP-buffer size.

The Parameter Tuner gets the different optimal values and generates overall optimal values. For instance, it makes sure that the disk I/O block size is equal to or a multiple of the TCP-buffer size.

The Tuning Infrastructure has the knowledge to perform protocol-specific tuning. For instance GridFTP takes as input only a single I/O block size, but the source and destination machines may have different optimal I/O block sizes. For such cases, the tuning finds the I/O block size which is optimal for both of them. The incremental block size measured by the disk profiler is used for this.

Finally, the parameter tuner creates a library of network links, protocols and tuned parameters. A part of the library showing the parameters for gridftp for the link from slic04.sdsc.edu to quest2.ncsa.uiuc.edu is shown below

```
[
  link = "slic04.sdsc.edu - quest2.ncsa.uiuc.edu";
  protocol = "gsiftp";
  bs        = 4096KB;     //block size
  tcp_bs    = 1024KB;    //tcp buffer size
  p         = 4;         //parallelism
]
```

An instance of the parameter tuner is run for every pair of nodes involved in the transfer and it can be executed on any computation node.

Network Interface Statistics Generation Tool. We developed a tool to generate the network interface statistics, which helps us find the actual bits flowing on the wire. It also shows the number of packets and the number of lost and dropped packets. It runs only on Linux and picks up the information from /proc. It essentially samples the kernel network counters. The sampling interval is tunable and we recommend setting it between 5 and 30 seconds. The lower value is to reduce the overhead and the upper value is because the counters are signed 32-bit and can overflow in 32 seconds on a gigabit link. The tool detects overflow and corrects the statistics appropriately. The statistics are very useful in debugging problems and identifying bottlenecks and network misconfiguration.

Statistics Collection Tool. We developed this tool to collect the network interface and data transfer statistics from the different nodes and aggregate them in a common place. For the data transfer statistics, if the data transfer tool generates statistics, we use them, otherwise we use coarse-grained statistics generated by Stork. The DiskRouter tool [13] generates extensive statistics. We run the statistics collection tool periodically. The tool is sophisticated enough to transfer only incremental updates. It also performs the role of logrotate for the statistics log.

Data Exploration and Visualization(DEVise). DEVise [18] is a data exploration system that allows users to easily develop, browse, and share visual presentations of large tabular datasets (possibly containing or referencing multimedia objects) from several sources. We used DEVise to visualize the data. We found that it is easy to locate problems or potential problems from visualization instead of looking through log files. Data visualization in DEVise consists of the following steps

1. Creation of Schemas.
2. Creation of tables with each table described by an appropriate schema.
3. Association of a data-source(file) with each table.
4. Creation of a session file describing the fields from tables to be displayed on each of the axes and also color and other information.

DEVise visualization can be made accessible via the web. The visualizations are interactive allowing users to zoom in to areas of interest and zoom out to see the overall pattern.

Data Processing Tool. We developed this tool to process the different statistics into DEVise table data. This tool has to be run after the data collection tool. The data processing tool also processes incremental data and appends them to existing DEVise data-sources.

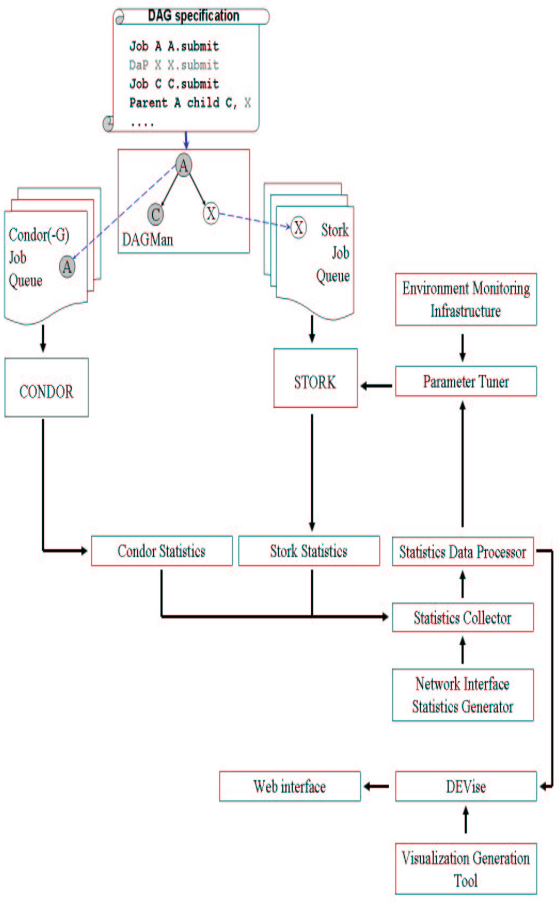

Figure 3. The Framework. ***This shows how the different parts of the framework come together.***

Visualization Generation Tool. This tool generates an appropriate visualization based on the number of nodes and the topology. Visualization generation needs to be done only once for a given configuration. In our framework, we run the visualization generation tool after the collection and processing of initial statistics. For the incremental statistics, processed data is appended to existing DEVise data-sources. We use a DEVise feature whereby the visualization automatically gets updated when the data-source is updated.

Figure 3 shows how all of these pieces com together in our framework. A DAG describing the data movement and computation is submitted to DAGMan. DAGMan then submits computational jobs to Condor, and data placement jobs to Stork. The Environment Monitoring Infrastructure feeds the environment data to the Parameter Tuner which generates optimal parameter values. The optimal parameter values are fed to Stork which uses them to tune the data transfers. The Network Interface Statistics Generator generates network interface statistics. The Statistics Collector collects the different statistics and feeds them to the Data Processor which converts the statistics to DEVise data and appends them to existing data-sources. The Visualization Generation Tool generates the visualization and publishes them on a web-site. DEVise gets run on the server side on demand and is responsible for displaying the visualization and updating it when the data-source gets updated.

4. Experiment

NCSA scientists wanted to perform certain processing on the Digital Palomar Sky Survey(DPOSS) [6] image data residing on SRB [2] mass storage system at SDSC in California. The total data size was around 3 TB (2611 files of 1.1 GB each). NCSA located in Illinois has its own mass-storage system called UniTree [3]. Since there was no direct interface between SRB and UniTree at the time of the experiment, the only way to perform the data transfer between these two storage systems was to move the data via intermediate nodes. For this purpose, we designed two different configurations using our framework to transfer the data. A metric of importance to use is the end-to-end transfer rate. We measured the end-to-end transfer rate of a file as the file size divided by the time interval between the instant the DAG for the transfer is submitted to DAGMan and the instant when the whole file reaches the destination and all intermediate copies of the file are deleted. The end-to-end transfer rate we report is the average end-to-end transfer rate for a set of files.

4.1. First Configuration

The first approach we employed was to set up a cache node at the NCSA site very close to the UniTree server. This approach allowed us to transfer the DPOSS data first from the SRB server to the NCSA cache node using the underlying protocol of SRB, and then from the NCSA cache node to UniTree server using the underlying protocol of UniTree. Figure 4 shows the topology of the network, bottleneck bandwidth and latencies.

The NCSA cache node had only 12 GB of local disk space for our use and we could store only 10 image files in that space. This implied that whenever we were done with a file at the cache node, we had to remove it from there to create space for the transfer of another file. Including the removal step of the file, the end-to-end transfer of

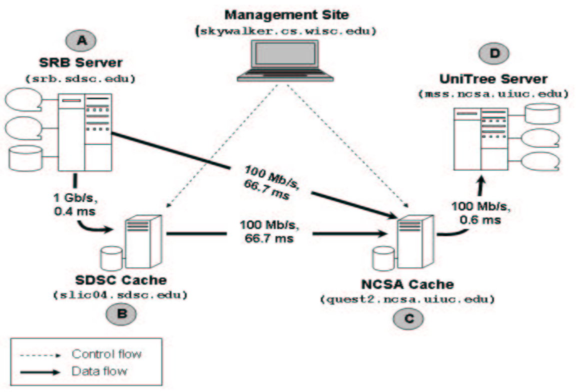

Figure 4. Network Topology. ***The topology of the network used in the transfers, with the bottleneck bandwidth and latency between each node.***

each file consisted of three basic steps, all of which we considered as real jobs to be submitted either to the Condor or Stork scheduling systems. Then all of these three node DAGs were joined together to form a giant DAG as shown in Figure 5, and the whole process was managed by DAGMan. The concurrency level k denotes the number of simultaneous file transfers taking place. Best throughput is got out of the mass storage system with a certain concurrency level. The SRB and UniTree servers had gigabit ethernet(1000 Mb/s) interface cards installed on them and the NCSA cache node had a fast ethernet(100 Mb/s) interface card installed on it. We found the bottleneck link to be the fast ethernet interface card on the NCSA cache node.

We got an end-to-end transfer rate of 40Mb/s from the SRB server to the UniTree server. We observed that the bottleneck was the transfers between the SRB server and the NCSA cache node. We found that the SRB protocol does not have sufficient parameters to tune for wide-area data transfers. So we decided to add another cache node at the SDSC site to regulate the wide area transfers.

4.2. Second Configuration

In the second configuration, we introduced another cache node to the system. This cache node was placed at the SDSC site, very close to the SRB server. In this case, the data is first transferred from the SRB server to the SDSC cache node using the underlying protocol of SRB, then from the SDSC cache node to the NCSA cache node using third-party GridFTP transfers, and finally from the NCSA cache node to the UniTree server using the underlying protocol of UniTree. The space limitations of the NCSA cache node applied to the SDSC cache node as well, which required careful cleanup of transferred files at both nodes. Including the

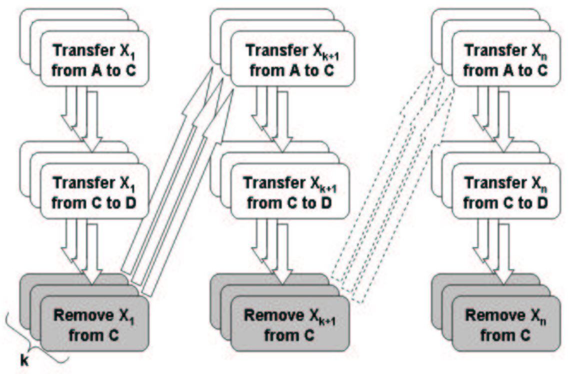
Transfer X_1 from A to C
Transfer X_{k+1} from A to C
Transfer X_n from A to C
Transfer X_1 from C to D
Transfer X_{k+1} from C to D
Transfer X_n from C to D
Remove X_1 from C
Remove X_{k+1} from C
Remove X_n from C
k

Figure 5. Transfer in 3 Steps. ***Three step DAGs are combined into a giant DAG to perform transfers in the first configuration. k is the concurrency level.***

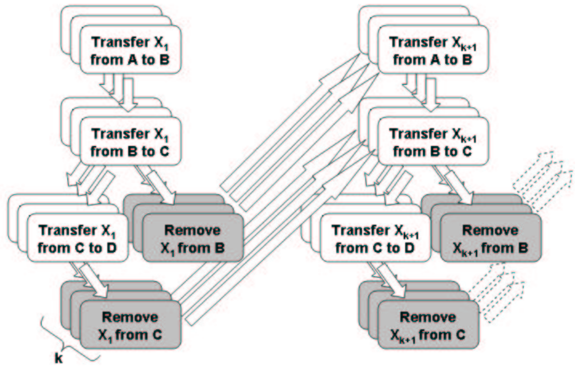
Transfer X_1 from A to B
Transfer X_{k+1} from A to B
Transfer X_1 from B to C
Transfer X_{k+1} from B to C
Transfer X_1 from C to D
Remove X_1 from B
Transfer X_{k+1} from C to D
Remove X_{k+1} from B
Remove X_1 from C
Remove X_{k+1} from C
k

Figure 6. Transfer in 5 Steps. ***Five step DAGs are combined into a giant DAG to perform the transfers in the second configuration, with concurrency level = k.***

cleanup steps, the end-to-end transfer of each file consisted of five basic steps as shown in Figure 6. Then all of these five node DAGs were joined together to form a giant DAG as in the previous pipeline. All of these jobs were executed by Condor and Stork systems.

The SDSC cache node had a gigabit ethernet interface card installed on it, but the link between the SDSC cache node and the NCSA cache node still had a bandwidth of 100Mb/s due to the fast ethernet interface of the NCSA cache node. Using this configuration, we got an end-to-end transfer rate of 25.6 Mb/s, and the link between the SDSC cache node and the NCSA cache node turned out to be the bottleneck. At this point, we had not implemented the auto-tuning infrastructure. We had done some intuitive hand-tuning before we started the experiment.

Having two cache nodes gave us flexibility. We now had the ability to use any protocol of choice to transfer data between the cache nodes. We used DiskRouter to transfer data between the two cache nodes and found that we got an end-to-end throughput of 47.6 Mb/s.

4.3. Automated Failure Recovery

The failure recovery mechanism in our framework works at multiple levels. Stork has a sophisticated failure recovery mechanism. Users can tell Stork to retry a transfer till it succeeds. They can also specify the number of retries. Stork logs the progress to persistent storage and resumes transfers even after crashes and reboots. DAGMan also has failure recovery mechanism and it too logs the progress to persistent storage and resumes after crashes and reboots. DAGMan after retrying for a specified number of times, creates a rescue DAG specifying the jobs which have not completed. This rescue DAG can again be fed to DAGMan to be retried.

During the course of the 3 Terabytes data movement, we encountered a wide variety of failures. At times, either the source or destination mass-storage systems stopped accepting new transfers. Such outages lasted about an hour on the average. In addition we had windows of scheduled maintenance activity. We also had wide-area network outages, some lasting a couple of minutes and others lasting longer. We also had software upgrades.

We found a need to insert a timeout on the data transfers. Occasionally we found that a data transfer command would hang. Most of the time, the problem occurred with third-party wide-area transfers. Once in a while, a third-party GridFTP transfer would hang. In the case of DiskRouter we found that the actual transfer completed but we were not notified of the completion. Because of these problems, we added a timeout-feature to Stork. When the timeout is set, Stork expects the data transfer to complete within the timeout. If it does not, Stork terminates the data transfer, performs cleanup and retries it. The results presented in this subsection were obtained using the second configuration with DiskRouter being used for data transfer between the cache nodes.

Figure 7 shows two outages. This visualization was generated by DEVise. The first outage happened because UniTree refused new transfers. It lasted for 40 minutes. At that point, two transfers to UniTree were in progress. The transfers completed before the timeout expired. The second outage lasted slightly more than one and a half hours. It was caused by a reconfiguration of the DiskRouter system. We would like to mention that in both of the cases, the data transfers resumed without human intervention and we no-

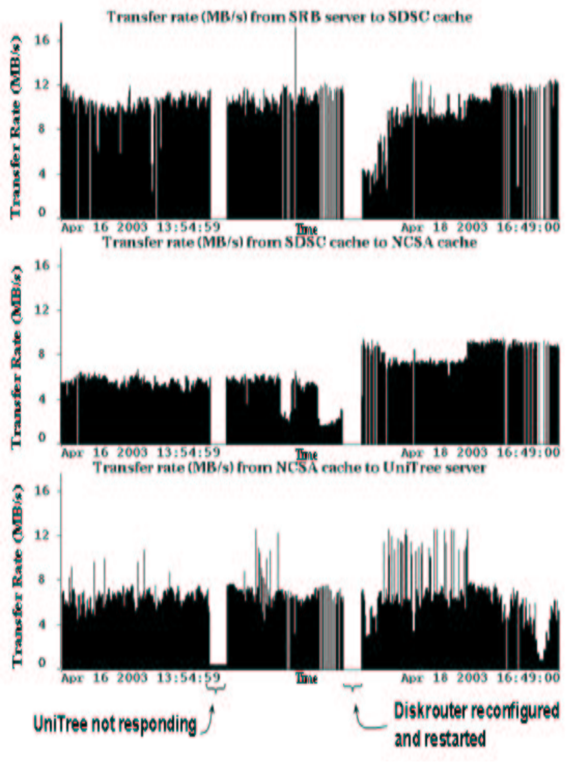

Figure 7. Automated Failure Recovery in case of Server Problem and Software Upgrade. *The transfers recovered automatically although first the UniTree server experiences some problems and then the DiskRouter servers running on the cache nodes get reconfigured and restarted.*

ticed them by looking at the visualization later and verified it with the log files.

In another case, first the SDSC cache machine was rebooted and then there was a UW CS network outage lasting a couple of hours. This disconnected the management/submit site from the execution sites. The framework automatically recovered from these two failures. Finally the DiskRouter server stopped responding for a couple of hours. The DiskRouter problem was partially caused by a network reconfiguration at StarLight hosting the DiskRouter server. Here again, our automatic failure recovery worked fine.

4.4. Testing the Run-time Protocol Auto-tuning

After implementing the run-time protocol auto-tuning, we wanted to see its effect on the wide-area transfer between the cache nodes. We submitted 500 requests to the Stork server to transfer the 1.1GB image files (total 550 GB) using GridFTP as the primary protocol. We used third-party globus-url-copy transfers without any tuning or without changing any of the default parameters.

Parameter	Before auto-tuning	After auto-tuning
parallelism	1 TCP stream	4 TCP streams
block size	1 MB	1 MB
tcp buffer size	64 KB	256 KB

Table 1. Network parameters for gridFTP before and after auto-tuning feature of Stork being turned on.

We turned off the auto-tuning feature of Stork at the beginning of the experiment intentionally. The average data transfer rate that globus-url-copy could get without any tuning was only 0.5 MB/s. The default network parameters used by globus-url-copy are shown in Table 1. After a while, we turned on the auto-tuning feature of Stork. Stork first obtained the optimal values for I/O block size, TCP buffer size and the number of parallel TCP streams from the parameter tuner. Then it applied these values to subsequent transfers. Figure 8 shows the increase in the performance after the auto-tuning feature is turned on. We got a speedup of close to 20 times compared to transfers without tuning. The auto-tuning mechanism performed better than our primitive hand-tuning. Here again the visualization was generated by DEVise and the statistics were collected and processed by the framework tools.

5. Future Work

We want to add a feature to Stork to differentiate between transient and permanent errors. For instance, a source file not existing is a permanent error, whereas the destination temporarily refusing to accept new transfers, is a transient error. We also plan to add a trigger mechanism to Stork, so that it can notify the designated person on encountering permanent errors. We are planning to use our framework for CMS data movement.

6. Conclusion

We have designed and implemented a framework that makes it easy to move data between heterogeneous systems. The framework has environment monitoring and tuning capability which significantly boosts the performance of data transfers. In our test case, the performance boost was 20 times. The framework is highly resilient and has features to cope with a variety of network, storage system and software failures.

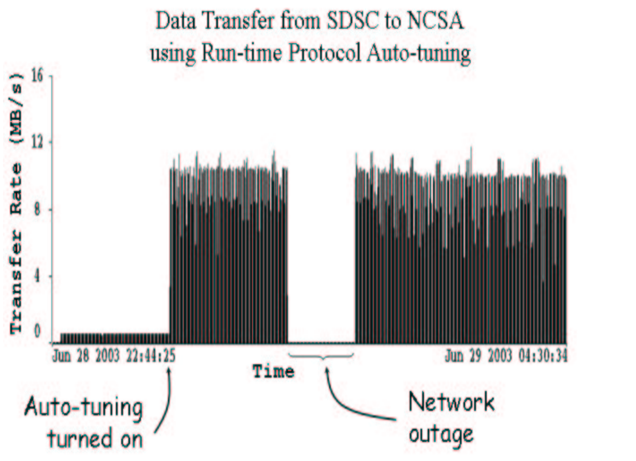

Figure 8. Run-time Protocol Auto-tuning. ***Stork starts the transfers using the GridFTP protocol with auto-tuning turned off intentionally. Then we turn the auto-tuning on, and the performance increases drastically.***

Through a real-life data transfer involving thousands of large files, we have shown that our framework works and is resilient to storage system, network, and software failures. We present our framework as a viable alternative to dumping data to tapes and FedExing them or writing scripts and baby-sitting the scripts to deal with failures.

7. Acknowledgements

We would like to thank Robert J. Brunner, Michelle Butler and Jason Alt from NCSA; Philip Papadopoulos, Mason J. Katz and George Kremenek from SDSC for the invaluable help in providing us access to their resources, support and feedback. We would like to also thank James Frey for helpful discussion and comments on the paper.

References

[1] B. Allcock, J. Bester, J. Bresnahan, A. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, and S. Tuecke. Secure, efficient data transport and replica management for high-performance data-intensive computing. In *IEEE Mass Storage Conference*, San Diego, CA, April 2001.

[2] C. Baru, R. Moore, A. Rajasekar, and M. Wan. The SDSC Storage Resource Broker. In *Proceedings of CASCON*, Toronto, Canada, 1998.

[3] M. Butler, R. Pennington, and J. A. Terstriep. Mass Storage at NCSA: SGI DMF and HP UniTree. In *Proceedings of 40th Cray User Group Conference*, 1998.

[4] CMS. The Compact Muon Solenoid Project. http://cmsinfo.cern.ch/.

[5] Condor. The Directed Acyclic Graph Manager. http://www.cs.wisc.edu/condor/dagman, 2003.

[6] S. G. Djorgovski, R. R. Gal, S. C. Odewahn, R. R. de Carvalho, R. Brunner, G. Longo, and R. Scaramella. The Palomar Digital Sky Survey (DPOSS). *Wide Field Surveys in Cosmology*, 1988.

[7] W. Feng. High Performance Transport Protocols. Los Alamos National Laboratory, 2003.

[8] I. Foster and C. Kesselmann. Globus: A Toolkit-Based Grid Architecture. In *The Grid: Blueprints for a New Computing Infrastructure*, pages 259–278, Morgan Kaufmann, 1999.

[9] J. Frey, T. Tannenbaum, I. Foster, and S. Tuecke. Condor-G: A Computation Management Agent for Multi-Institutional Grids. In *Tenth IEEE Symp. on High Performance Distributed Computing*, San Francisco, CA, August 2001.

[10] R. Henderson and D. Tweten. Portable Batch System: External Reference Specification, 1996.

[11] IBM. Using and Administering IBM LoadLeveler. IBM Corporation SC23-3989, 1996.

[12] D. Koester. Demonstrating the TeraGrid - A Distributed Supercomputer Machine Room. *The Edge, The MITRE Advanced Technology Newsletter*, 6(2), 2002.

[13] G. Kola and M. Livny. Diskrouter: A Flexible Infrastructure for High Performance Large Scale Data Transfers. Technical Report CS-TR-2003-1484, University of Wisconsin, 2003.

[14] S. Koranda and B. Moe. Lightweight Data Replicator. http://www.lsc-group.phys.uwm.edu/lscdatagrid/LDR.

[15] T. Kosar and M. Livny. Scheduling Data Placement Activities in the Grid. Technical Report CS-TR-2003-1483, University of Wisconsin, 2003.

[16] LIGO. Laser Interferometer Gravitational Wave Observatory. http://www.ligo.caltech.edu/, 2003.

[17] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor - A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, pages 104–111, 1988.

[18] M. Livny, R. Ramakrishnanand, K. Beyerand, G. Chenand, D. Donjerkovicand, S. Lawandeand, J. Myllymaki, and K. Wenger. Devise: Integrated querying and visual exploration of large datasets. In *Proceedings of ACM SIGMOD*, May 1997.

[19] R. Maddurri and B. Allcock. Reliable File Transfer Service. http://www-unix.mcs.anl.gov/ madduri/main.html, 2003.

[20] B. Sagal. Grid Computing: The European DataGrid Project. In *IEEE Nuclear Science Symposium and Medical Imaging Conference*, Lyon, France, October 2000.

[21] D. Thain, J. Basney, S. Son, and M. Livny. The kangaroo approach to data movement on the grid. In *Proceedings of the Tenth IEEE Symposium on High Performance Distributed Computing*, San Francisco, California, August 2001.

[22] S. Zhou. LSF: Load Sharing in Large-Scale Heterogeneous Distributed Systems. In *Proc. of Workshop on Cluster Computing*, 1992.