Data Pipelines: Enabling Large Scale Multi-Protocol Data Transfers

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

Collaborating users need to move terabytes of data among their sites, often involving multiple protocols. This process is very fragile and involves considerable human involvement to deal with failures. In this work, we propose data pipelines, an automated system for transferring data among collaborating sites. It speaks multiple protocols, has sophisticated flow control and recovers automatically from network, storage system, software and hardware failures. We successfully used data pipelines to transfer three terabytes of DPOSS data from SRB mass storage server at San Diego Supercomputing Center to UniTree mass storage at NCSA. The whole process did not require any human intervention and the data pipeline recovered automatically from various network, storage system, software and hardware failures.

Categories and Subject Descriptors

D.1.3 [**Software**]: Programming Techniques—*Concurrent Programming*

General Terms

Performance, Design, Reliability, Experimentation

Keywords

Data Pipelines, distributed systems, grid, replication, bulk data transfers, fault-tolerance, mass storage systems

1. INTRODUCTION

Grid computing [8] has enabled researchers to collaborate more effectively by sharing computing resources. Many fields including astronomy, genetics, biomedicine and geology have to transfer large amounts of data among their collaborating organization. Each organization either developed or decided to use one particular storage system. For example, NCSA uses UniTree [3], SDSC uses SRB [2], LBNL uses HPSS [14] and Fermi uses Enstore [7] mass storage systems.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
MGC'04, October 18, 2004, Toronto, Canada.
Copyright 2004 ACM 0-89791-88-6/97/05 ...$5.00.

Collaborating researchers spanning across the different organizations need to transfer data among different storage systems and most of the time have to speak multiple protocols. While reliably transferring large amounts of data over the wide-area is in itself difficult, the additional complexity of speaking multiple protocols makes the process more fragile. The current approach has been to use an operator at each site to monitor the transfers.

We propose data pipelines, a system to automate data transfer among multiple sites. Similar to water pipelines that automatically regulate water flow depending on the consumption at end-points, the data flow is regulated by the consumption rate/bandwidth at end-points. It speaks multiple protocols, is highly resilient and can recover from network, storage-server, software and hardware failures.

In this work, we show the design of data pipelines, discuss the implementation and highlight the functionality that it provides. We successfully used it to replicate the three terabytes DPOSS [5] dataset from SRB mass storage system at SDSC to UniTree mass storage system at NCSA.

2. MOTIVATION

Researchers wanting to share terabytes datasets with their collaborators are finding it difficult to manage the process. While the underlying network capacity has grown enough to make this possible, the management part of the process has not matured.

Different organizations use different data access protocols. This makes it a challenge to move data between these organizations. While one approach may be to force all of them to agree on one protocol, we believe that it may not be always possible. Further, supporting a new protocol in a storage server incurs considerable development cost and may take time to reach the stability of existing protocol. For instance, SRB server had a GridFTP interface close to working and later dropped it because of stability issues [21].

Large data transfers have a higher likelihood of noticing wide-area network outages because they span a longer time span. Even though most wide-area network outages are of short duration, most data transfer protocols cannot recover automatically from these failures. Short-duration failures do not affect interactive web users much because in most cases, pressing the reload browser button a couple of times may be sufficient to recover from the outage.

Hardware and software failures may occur on the storage server, client machine and intermediate nodes used for staging and protocol translation. Periodic maintenance and emergency maintenance to fix vulnerabilities may appear as failures from the end-to-end transfer point of view, as a large-scale data transfer may span across multiple maintenance periods. These system failures result in user visible failures and require user intervention to restart the failed transfers. This may require considerable work on the user's part if

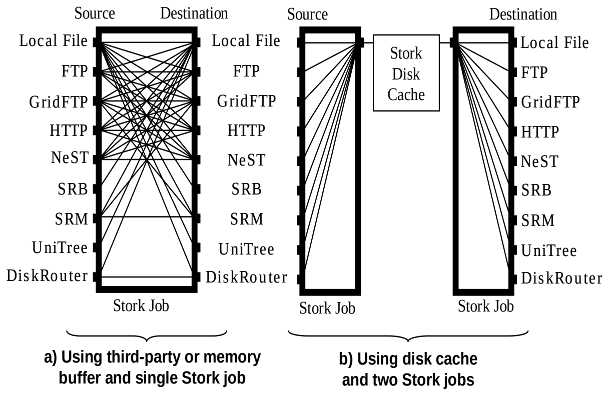
Source
Destination
Local File
FTP
GridFTP
HTTP
NeST
SRB
SRM
UniTree
DiskRouter
Stork Job
Stork Disk Cache
a) Using third-party or memory buffer and single Stork job
b) Using disk cache and two Stork jobs

Figure 1: Shows Stork protocol translation using memory and disk caches

the transfer involves a dataset with several thousands of files. Most users just want the system to recover from the failures.

Paxson [19] found that TCP checksum is not sufficient when moving large amounts of data. By the end-to-end argument [20], the only option is to compute end-to-end checksum and verify that they match. Since some storage servers do not allow computation, calculating the checksum may involve transfer to a local node and calculating the checksum there. While doing this, the system should ensure that this local node is not corrupting the data. Users want a flexible system that can accomplish this without requiring much effort on their part.

Many data access protocols that organizations use internally do not work well over wide area. For instance, SRB protocol has issues with certain wide area transfers, as it does not allow tuning of TCP window size. Similarly, many mass storage protocols are optimized for local access and they have difficulty coping with the variety of wide-area failures. Some of them just hang complicating the life of the user/operator, as he now has to find out transfers that have hung.

Many operators have scripts that attempt to automate the data transfer. While scripts work to a certain extent in the presence of a common interface, they have difficulty when transferring data using multiple protocols. This is because end-to-end flow control in a multi-hop data transfer is difficult to perform in scripts. They have to use an intermediate node and carefully manage the space on that node. At times, the destination may have a failure and even when the destination recovers from the failure, the script may end up having other failures because of full-disks at intermediate nodes or end up having sub-optimal performance.

3. METHODOLOGY

To handle failures, we make data placement a full-fledged job. Data placement encompasses data transfer, staging, replication, data positioning, space allocation and de-allocation. We queue, schedule and manage data placement jobs just like computational jobs. To accomplish this, we have developed Stork data placement scheduler [13].

Making data placement as a full-fledged job considerably improves failure handling. We can build on job-level failure recovery techniques developed for computation. For instance, we can provide alternate job failure-recovery where we execute an alternate job on encountering a failure. For data transfer, the alternate job may determine the correctly transferred parts of the file and resume the transfer from previous saved state.

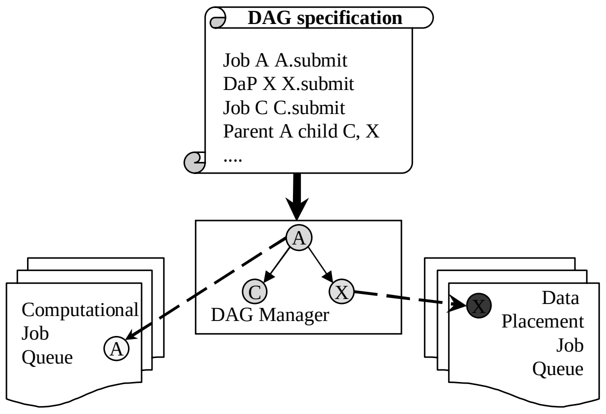

Figure 2: Prototype Model. ***A DAG specification file consisting of both computational and data placement jobs is submitted to DAGMan. DAGMan then submits computational jobs to Condor, and data placement jobs to Stork.***

We have added alternate protocol support to Stork for improving failure recovery. Here users can specify an ordered list of protocols. On encountering a failure, Stork switches to the next protocol in the list. This is also useful when there are new fast protocols that have not yet stabilized. With alternate protocol support, Stork uses the new protocol when it works and switches to the slower more reliable protocol on failure. Users can benefit from newer protocols without having to worry about manual failure recovery.

To handle data transfer with multiple protocols, we have added a stork module that can perform protocol translation using in-memory buffer or disk buffer. Figure 1 shows this. While using disk buffer, we allocate and de-allocate space to prevent over-committing disk and to avoid the resultant failures.

Performing checksum requires more flexibility, especially since the storage server may not support checksum. To handle this, we let users execute a direct acyclic graph of computational and data placement jobs. Thus, users can transfer data from source to destination via an intermediate node for protocol translation and if the destination server does not support checksum, copy the data to a local node and compute the checksum there. Permitting users to run arbitrary computation enables users use sophisticated techniques like network encoding [15] whereby they can reconstruct the data from certain set of blocks .

We use Condor [17]/Condor-G [10] as the computational scheduler. Condor-G uses the Globus [9] toolkit functionality to submit computation jobs to any grid-enabled scheduling system. To perform the management of the DAGs, we employed the Directed Acyclic Graph Manager (DAGMan) [4, 23], 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. We have enhanced DAGMan with data placement support and our DAGMan submits the data placement job to Condor and the data placement job to Stork. Figure 2 shows this process.

Due to the storage limitations of intermediate nodes in a data-pipeline, we need to remove the files from intermediate nodes after they have reached the next stage of the pipeline. The DAG model is flexible enough to accommodate that.

Some transfers may just hang for a long time due to problems

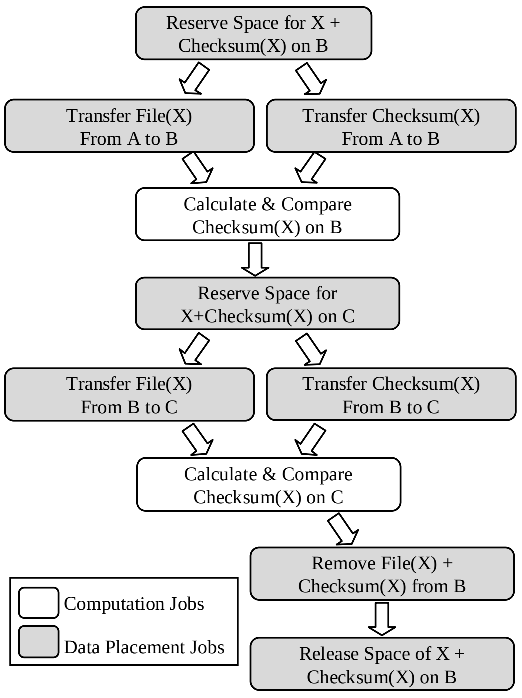
Reserve Space for X + Checksum(X) on B
Transfer File(X) From A to B
Transfer Checksum(X) From A to B
Calculate & Compare Checksum(X) on B
Reserve Space for X+Checksum(X) on C
Transfer File(X) From B to C
Transfer Checksum(X) From B to C
Calculate & Compare Checksum(X) on C
Remove File(X) + Checksum(X) from B
Release Space of X + Checksum(X) on B
Computation Jobs
Data Placement Jobs

Figure 3: All steps in the pipeline are represented as full-fledged jobs, and the dependencies between jobs are represented as directed arcs.

in the system, or bugs in the underlying protocol implementation. To handle this, the system should let users specify timeouts. If a transfer takes longer than a certain amount of time, the system should terminate and restart them. In our scheme, either DAGMan or the scheduler can implement this ”kill-and-restart” mechanism. We implemented the kill-and-restart mechanism in Stork. Users can specify the time-out statically by assuming a certain minimum transfer rate or an agent can specify it dynamically taking into account the performance of previous transfers.

Figure 3 shows the steps involved in transferring a single file from node A to node C using an intermediate nodes B for staging and protocol translation. All of the steps including data transfer, space allocation and de-allocation, removal of temporary files and checksum computations are full-fledged jobs. Depending on the characteristics of the jobs, DAGMan submits to either a computational or a data placement scheduler. Directed arcs represent dependencies between jobs. For the transfer of multiple files, we merge these DAGs into a giant DAG with dependencies to limit number of files transferred concurrently and to prevent over-committing the disk space on the staging nodes.

4. EVALUATION

National Center for Supercomputing Applications (NCSA) scientists wanted to replicate the Digital Palomar Sky Survey (DPOSS) [5] image data residing on SRB mass storage system at San Diego Supercomputing Center (SDSC) in California to UniTree mass-storage system at NCSA, Illinois to enable them to perform later processing. The total data size was around three terabytes (2611 files of 1.1 GB each). 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 use an intermediate node to perform protocol translation. Hoping

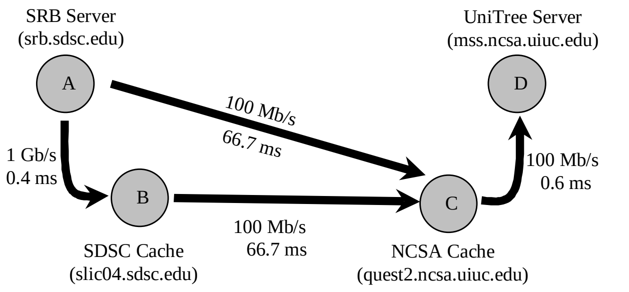

Figure 4: The topology of the network used in the transfers, with the bottleneck bandwidth and latency between each node.

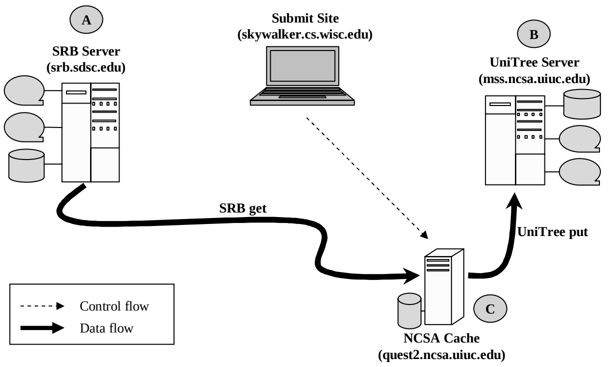

Figure 5: Building a data-pipeline with one intermediate node may be sufficient for data transfer between two heterogeneous storage systems.

to avoid hiring an operator, NCSA astronomers decided to try our data pipelines. For this purpose, we designed three different data pipelines to transfer the data and to evaluate our system.

We had access to two cache nodes: one at SDSC (slic04.sdsc.edu) and other at NCSA (quest2.ncsa.uiuc.edu). The SRB, UniTree servers and SRB cache node had gigabit Ethernet(1000 Mb/s) interface and the NCSA cache node had a fast Ethernet(100 Mb/s) interface. The local area network at SRB was a gigabit and the wide-area network was 622 Mbps ATM shared among all users. The bottleneck link was the fast Ethernet interface card on the NCSA cache node. Figure 4 shows the topology of the network, bottleneck bandwidth and latencies.

4.1 First Data Pipeline Configuration

In the first data pipeline, we used the NCSA cache node, quest2.ncsa.uiuc.edu, to perform protocol translation. We transferred the DPOSS data from the SRB server to the NCSA cache node using the underlying SRB protocol and from the NCSA cache node to UniTree server using UniTree mssftp protocol. Figure 5 shows this pipeline configuration.

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 required careful space management and we had to remove a file immediately after transferring it to UniTree to create space for the transfer of next file.

We got an end-to-end transfer rate of 40Mb/s from the SRB server to the UniTree server. We calculated the end-to-end transfer rate by dividing the data transferred over a two-day period by the total time taken (2 days) and repeating the process three times interleaved with other pipeline configuration in random order and

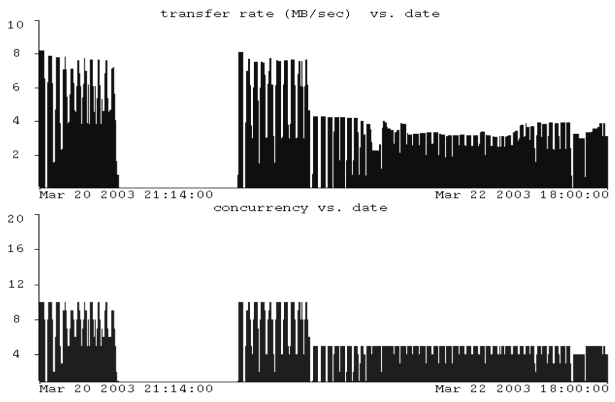

Figure 6: Throughput and concurrency of the transfers between SDSC and NCSA using the first pipeline configuration.

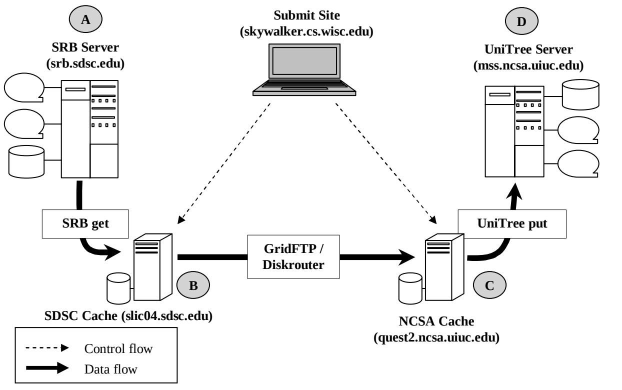

Figure 7: Building a data-pipeline with two intermediate nodes, one close to the source and one close to the destination, may provide additional functionality and increase performance.

averaging. We also applied statistical methods and verified that day effect was statistically insignificant at 90-percentage level. We observed that the bottleneck was the transfers between the SRB server and the NCSA cache node. As SRB protocol did not allow us to tune TCP windows size forcing us to increase concurrency to achieve a similar affect, we decided to add another cache node at the SDSC site to regulate the wide area transfers.

Figure 6 is a snapshot showing the throughput and concurrency level of the system over time. There was a six-hour UniTree maintenance period during which the transfers stopped and than resumed. At some point, the SRB server started refusing new connections. The pipeline reduced the concurrency level automatically to decrease the load on the SRB server.

4.2 Second Data Pipeline Configuration

In the second pipeline configuration, we used both SDSC and NCSA cache nodes. We transfer the data from the SRB server to the SDSC cache node using the SRB protocol, 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 UniTree mssftp protocol. Figure 7 shows this pipeline configuration. SDSC cache node also had space limitations requiring careful cleanup of transferred files at both cache nodes.

While this step may seem like an additional copy, we did not have source checksum. By transferring data to a local node, we were able to calculate the checksum and verify it at the destination. 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 was the bottleneck.

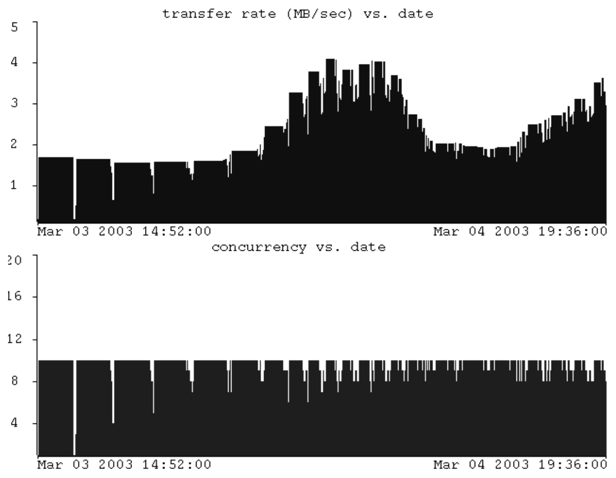

Figure 8: Throughput and concurrency of the transfers between SDSC and NCSA using the second pipeline configuration.

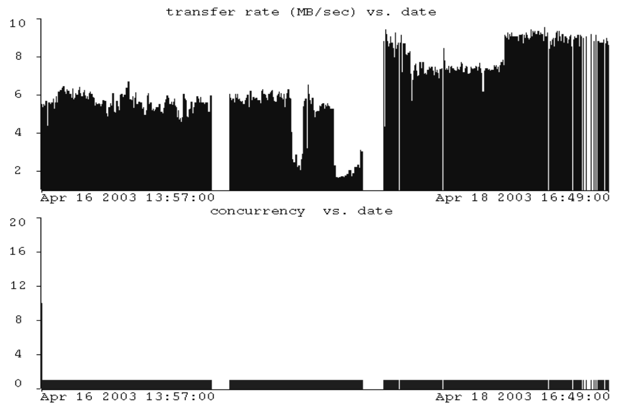

Figure 9: Throughput and concurrency of the transfers between SDSC and NCSA using the third pipeline configuration

Figure 8 shows the throughput and concurrency level of the system over time. For optimal performance, we wanted a concurrency level of ten and the system maintained it. The fluctuations in the throughput are due to changing network conditions and GridFTP not aggressively utilizing the full bandwidth.

4.3 Third Data Pipeline Configuration

The third data pipeline configuration was almost the same as the second one, except that we replaced third-party GridFTP transfers between the SDSC cache node and the NCSA cache node, which were the bottleneck, with third-party DiskRouter [11] transfers.

This time we got an end-to-end throughput of 47.6 Mb/s. DiskRouter works best with a single transfer and it effectively utilizes the available bandwidth with a single transfer. Figure 9 shows two failures that occurred during the transfers. The first one was a UniTree server problem, and the second one was reconfiguration of DiskRouter that improved its performance. The system recovered automatically in both cases.

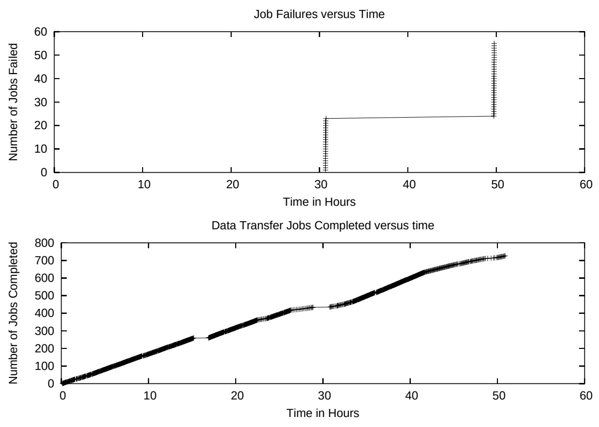

Figure 10: Shows automated failure recovery in third pipeline configuration.

4.4 Comparison of Pipeline Configurations

Comparison of performance of pipeline 1 and pipeline 2 shows the penalty associated with adding a node to the pipeline. Third pipeline configuration, which is similar to the second with GridFTP replaced by DiskRouter, performs better than first and second because DiskRouter dynamically tunes the socket-buffer size and the number of sockets according to the network conditions and it uses buffering at Starlight network access point to aid in the data transfers.

Carefully tuning the I/O and socket buffer sizes in GridFTP would substantially improve its performance and we can integrate an agent that can do it dynamically into the data-pipeline. This also shows that running wide-area optimized protocols between cache nodes can improve performance enough to offset the penalty of an additional node.

Adding extra nodes can result in increased flexibility. For instance, with pipeline 2 and 3, we can compute source checksum and verify it at the destination. If the source checksum does not exist, as is the case with the DPOSS data, we need to compute it on a local node on the source network. To verify that this node is not corrupting the data, we can apply statistical techniques, transfer some data to other local nodes, and verify that the checksums generated on those nodes match with those generated on the stage node. Finally, if the destination also does not support checksum, as is the case with UniTree, we need to download the data to some other local node on the destination network and compute the checksum there and verify it with the source checksum. We can accomplish this easily using the DAG.

The pipelines mentioned here are just highlights of what is possible with data pipelines. The pipelines are inherently flexible and we have been able to build a distribution network to distribute the DPOSS dataset to compute nodes at NCSA, Starlight and UW-Madison.

4.5 Automated Failure Recovery

The most difficult part in operating data-pipelines is handling failures in an automated manner. During the course of the three Terabytes data movement, we had a wide variety of failures.

At times, either the source or the 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. While

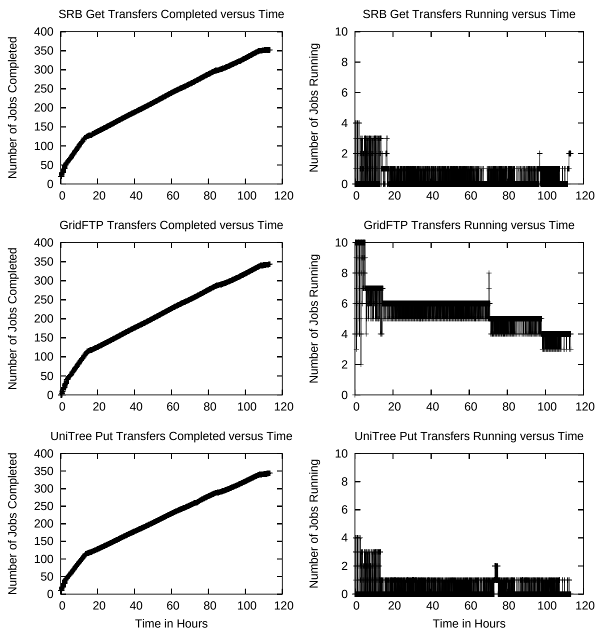

Figure 11: Shows automated failure and flow control in the second pipeline configuration

the pipeline was in operation, we had software upgrades. We also 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. Occasionally, a third-party GridFTP transfer would hang. In the case of DiskRouter, we found that the actual transfer completed but DiskRouter did not notify us of the completion. Because of these problems, we set a timeout for the transfers. If any transfer does not complete within the timeout, Stork terminates it, performs the necessary cleanup and restarts the transfer.

Figure 10 shows how the third data pipeline configuration automatically recovers from two sets of failures. At around 15-17 hours, SRB transfers almost hung. They look a long time, around one hour and 40 minutes, but lesser than the two-hour time-out for a transfer. This could have been due to some maintenance or some other higher priority job using all the tape drives at the mass storage. The transfers did not fail but completed after that period, so it does not appear as failure. Around 30 hours, there was short wide-area network outage. This resulted in DiskRouter failures. Another wide-area network outage at around 50 hours resulted in the second set of failures. The data pipeline recovered automatically from all these failures.

Figure 11 shows how the pipeline adapts the flow control on the fly. Around 4 hours, GridFTP encounters some wide-area failures and the pipeline lowers the number of concurrent transfers to seven. Close to 20 hours, SRB refuses new connection and the pipeline responds by trying to maintain a single connection. This affects the next hop and the number of concurrent GridFTP transfers drops to six. After that, UniTree accepts more connections and then slows down and this causes GridFTP to drop the number of concurrent transfers to five because of space limitations at the NCSA cache node. The next UniTree failure, at close to 100 hours, makes GridFTP drop the number of concurrent connections to four. The system was working through all these and users did not notice any failures. The end-to-end transfer rate observed by seeing the number of UniTree put transfers that completed show how well behaved the system is. Even though different flow control issues take place, the system is quite effective at maintaining the throughput.

5. RELATED WORK

Allcock et al. [1] introduce the GridFTP protocol and Replica Catalog and discuss how they enable secure and efficient data transfer and data replication. Reliable File Transfer Service (RFT) [18] transfers byte streams reliably. It handles a wide variety of problems like dropped connections, machine reboots, and temporary network outages automatically via retrying. Kangaroo [22] 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 move data between systems supporting their interface, but they cannot move data between heterogeneous storage systems lacking a common interface. All of them support only point-to-point transfers whereas data pipelines automatically manage multi-hop transfers.

Feng [6] mentions a case where visualization scientists at Los Alamos National Lab 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.

Lightweight Data Replicator (LDR) [12] can replicate data sets to the member sites of a Virtual Organization or Data Grid. Koranda et al. developed it using Globus tools for replicating LIGO [16] data. In its present form, LDR expects the use of a single data transport protocol (GridFTP). Our work is more general in nature and in addition to being able to manage point-to-point transfers like LDR, it can manage multi-hop data transfers between systems that do not support a common data transport protocol.

6. CONCLUSION

In this paper, we have proposed data-pipelines, an automated system for managing multi-hop data transfers. The system is flexible and allows running computation on the data as well. It is resilient to failures and can recover automatically from a variety of network, storage system, software and hardware failures. Through a real-life data transfer involving thousands of large files, we have shown that data pipeline works and is able to handle failures effectively. We present data pipelines as a viable alternative to dumping data to tapes and fedexing them or writing scripts and baby-sitting the scripts to deal with failures. We have shown that adding additional nodes does not necessarily decrease the end-to-end performance of the system and may in fact increase flexibility and improve performance if done properly.

We are planning to build automatic tuning capability into the system. Specifically we would like to add a feature to dynamically determine the optimal concurrency level for the different protocols. We are also considering building functionality into the system to dynamically choose the optimal pipeline configuration. We are planning on a network-monitoring infrastructure which would allow us to choose where to place the DiskRouter nodes and how many to place so that we get the best performance.

7. ACKNOWLEDGMENTS

We would like to thank Robert Brunner from NCSA and Philip Papadopoulos from SDSC for collaborating with us, letting us use their resources and making it possible to evaluate our data pipelines on a real life dataset replication.

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

[5] 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.

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

[7] FNAL. Enstore mass storage system. http://hppc.fnal.gov/enstore/, 2003.

[8] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of Supercomputing Applications*, 2001.

[9] 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.

[10] 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.

[11] 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.

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

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

[14] LBNL. High Performance Storage System. http://www.nersc.gov/nusers/resources/HPSS/, 2003.

[15] S.-Y. R. Li, R. W. Yeung, and N. Cai. Linear network coding. *IEEE transactions on information theory,*, 49(2):371–380, February 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] R. Madduri and B. Allcock. Reliable File Transfer Service. http://www-unix.mcs.anl.gov/ madduri/main.html, 2003.

[19] V. Paxson. End-to-End Internet Packet Dynamics. *IEEE/ACM Transactions on Networking*, 7(3):277–292, 1999.

[20] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, nov 1984.

[21] W. Schroeder. [srb-chat] srb and gridftp. https://lists.sdsc.edu/pipermail/srb-chat/2004-June/001077.html, 2004.

[22] 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.

[23] D. Thain, T. Tannenbaum, and M. Livny. Condor and the Grid. In *Grid Computing: Making the Global Infrastructure a Reality.*, Fran Berman and Geoffrey Fox and Tony Hey, editors. John Wiley and Sons Inc., 2002.