# Stork: Making Data Placement a First Class Citizen in the Grid

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

## Abstract

*Todays scientific applications have huge data requirements which continue to increase drastically every year. These data are generally accessed by many users from all across the globe. This implies a major necessity to move huge amounts of data around wide area networks to complete the computation cycle, which brings with it the problem of efficient and reliable data placement. The current approach to solve this problem of data placement is either doing it manually, or employing simple scripts which do not have any automation or fault tolerance capabilities. Our goal is to make data placement activities first class citizens in the Grid just like the computational jobs. They will be queued, scheduled, monitored, managed, and even check-pointed. More importantly, it will be made sure that they complete successfully and without any human interaction. We also believe that data placement jobs should be treated differently from computational jobs, since they may have different semantics and different characteristics. For this purpose, we have developed Stork, a scheduler for data placement activities in the Grid.*

## 1. Introduction

As the Grid [10] evolves, the data requirements of scientific applications increase drastically. Just a couple of years ago, the data requirements for an average scientific application were measured in Terabytes, whereas today we use Petabytes to measure them. Moreover, these data requirements continue to increase rapidly every year. A good example for this is the Compact Muon Solenoid (CMS) [5] project, a high energy physics project participating in the Grid Physics Network (GriPhyN). According to the Particle Physics Data Grid (PPDG) deliverables to CMS, the data volume of CMS, which is currently a couple of Terabytes per year, is expected to subsequently increase rapidly, so that the accumulated data volume will reach 1 Exabyte (1 million Terabytes) by around 2015 [19]. This is the data volume required by only one application, and there are many other data intensive applications from other projects with very similar data requirements, ranging from genomics to biomedical, and from metallurgy to cosmology.

The problem is not only the huge I/O needs of these data intensive applications, but also the number of users who will access the same datasets. For each of the projects, number of people who will be accessing the datasets range from 100s to 1000s. Furthermore, these users are not located at a single site, rather they are distributed all across the country, even the globe. So, there is a predominant necessity to move huge amounts of data around wide area networks to complete the computation cycle, which brings with it the problem of efficient and reliable data placement. Data need to be located, moved, staged, replicated, and cached; storage should be allocated and de-allocated for the data whenever necessary; and everything should be cleaned up when the user is done with the data.

Just as compute resources and network resources need to be carefully scheduled and managed, the scheduling of data placement activities all across the Grid is crucial, since the access to data has the potential to become the main bottleneck for data intensive applications. This is especially the case when most of the data is stored on tape storage systems, which slows down access to data even further due to the mechanical nature of these systems.

Currently, data placement activities in the Grid are performed either manually or by simple scripts. We can say that data placement activities are regarded as second class citizens of the computation-dominated Grid world. Our goal is to make data placement activities first class citizens in the Grid just like the computational jobs. They need to be queued, scheduled, monitored, managed, and even check-pointed.

We also believe that data placement jobs should be treated differently from computational jobs, since they may have different semantics and different characteristics. Existing computational job schedulers do not understand the semantics of data transfers well. For example, if the transfer of a large file fails, we may not want to simply restart the job and re-transfer the whole file. Rather, we may prefer to transfer only the remaining part of the file. Similarly, if a transfer using one protocol fails, we may want to try other protocols supported by the source and destination hosts to perform the transfer. A traditional computational job scheduler may not be able to handle these cases. For this purpose, data placement jobs and computational jobs should be differentiated. Data placement jobs should be submitted to a scheduler capable of scheduling and managing data placement jobs, and computational jobs should be submitted to a scheduler capable of scheduling and managing computational jobs. This will also help in separating computational and data placement jobs from each other and will give the users the ability to perform them asynchronously. For this purpose, we have developed Stork, a scheduler for data placement activities in the Grid.

## 2. Grid Data Placement Challenges

The Grid provides researchers with enourmous resources, but it also brings some challenges with it. In order to utilize Grid resources efficiently, researchers have to overcome these challenges first. Some of the data placement related challenges we are trying to solve with this work are below.

**Heterogeneous Resources.** The Grid is a heterogeneous environment in which many different storage systems, different data transfer middleware and protocols coexist. And it is a fundamental problem that the data required by an application might be stored in heterogeneous repositories. It is not an easy task to interact with all possible different storage systems to access the data. So there should be a negotiating system through which you can access all different kinds of storage systems, and also you can make use of all different underlying middleware and file transfer protocols.

**Hiding Failures from Applications.** The Grid brings failed network connections, performance variations during transfers, crashed clients, servers and storage systems with it. But generally the applications are not prepared to these kind of problems. Most of the applications assume perfect computational environments like failure-free network and storage devices, unlimited storage, availability of the data when the computation starts, and low latency. We cannot expect every application to consider all possible failures and performance variations in the system, and be prepared for them. Instead, we should be able to hide these from the application by a mediating system.

**Different Job Requirements.** Each job may have different policies and different priorities. Scheduling should be done according to the needs of each individual job. Global scheduling decisions should be able to be tailored according to the individual requirements of each job. Using only global policies may not be affective and efficient enough. The job description language used should be strong and flexible enough to support job level policies. And the job scheduler should be able to support and enforce these policies.

**Overloading Limited Resources.** The network and storage resources that an application has access to can be limited, and therefore they should be used efficiently. A common problem in distributed computing environments is that when all jobs submitted to remote sites start execution at the same time, they all start pulling data from their home storage systems (stage-in) concurrently. This can overload both network resources and the local disks of remote execution sites. It may also bring a load to the home storage systems from where the data is pulled.

One approach would be to pre-allocate both network and storage resources before using them. This approach works fine as long as the pre-allocation is supported by the resources being used, and also if the user knows when and how long the resources will be used by the application beforehand.

A more general solution would be to control the total number of transfers happening anytime between any given two sites. Most job schedulers can control the total number of jobs being submitted and executed at any given time, but this solution is not sufficient always and it is not the best solution in most cases either. The reason is that it does not do any overlapping of CPU and I/O, and causes the CPU to wait while I/O is being performed. Moreover, the problem gets more complex when all jobs complete and try to move their output data back to their home storage systems (stage-out). In this case stage-ins and stage-outs of different jobs may interfere, especially overloading the network resources more. An intelligent scheduling mechanism should be developed to control the number of stage-in and stage-outs from and to any specific storage systems anytime, and meanwhile do not cause any waste in CPU time.

## 3. Related Work

Visualization scientists at Los Alamos National Laboratory (LANL) found a solution for data placement by dumping data to tapes and sending them to Sandia National Laboratory (SNL) via Federal Express, because this was faster than electronically transmitting them via TCP over the 155 Mbps(OC-3) WAN backbone [8].

The Reliable File Transfer Service(RFT) [17] allows byte streams to be transferred in a reliable manner. RFT can handle wide variety of problems like dropped connections, machine reboots, and temporary network outages automatically via retrying. RFT is built on top of GridFTP [1], which is a secure and reliable data transfer protocol especially developed for high-bandwidth wide-area networks.

The Lightweight Data Replicator (LDR) [14] can replicate data sets to the member sites of a Virtual Organization or DataGrid. It was primarily developed for replicating LIGO [15] data, and it makes use of Globus [11] tools to transfer data. Its goal is to use the minimum collection of components necessary for fast and secure replication of data. Both RFT and LDR work only with a single data transport protocol, which is GridFTP.

There is ongoing effort to provide a unified interface to different storage systems by building Storage Resource Managers (SRMs) [22] on top of them. Currently, a couple of data storage systems, such as HPSS [21], Jasmin [3] and Enstore [9], support SRMs on top of them. SRMs can also manage distributed caches using "pinning of files". The SDSC Storage Resource Broker (SRB) [2] aims to provide a uniform interface for connecting to heterogeneous data resources and accessing replicated data sets. SRB uses a Metadata Catalog (MCAT) to provide a way to access data sets and resources based on their attributes rather than their names or physical locations.

Thain et. al. propose the Ethernet approach [23] to Grid Computing, in which they introduce a simple scripting language which can handle failures in a manner similar to exceptions in some languages. The Ethernet approach is not aware of the semantics of the jobs it is running, its duty is retrying any given job for a number of times in a fault tolerant manner. Kangaroo [24] tries to achieve high throughput by making opportunistic use of disk and network resources.

# 4. Stork Solutions to Grid Data Placement Problems

Stork provides solutions for many of the data placement problems encountered in the Grid environment.

## 4.1. Interaction with Higher Level Planners

Most of the applications in Grid require moving the input data for the job from a remote site to the execution site, executing the job, and then moving the output data from execution site to the same or another remote site. If the application does not want to take any risk of getting out of disk space at the execution site, it may also want to allocate space before transferring the input data there, and release the space after it moves out the output data from there.

We regard all of these these computational and data placement steps as real jobs and represent them as nodes in a Directed Acyclic Graph (DAG). The dependencies between them are represented as directed arcs, as shown in Figure 1.

Stork can interact with higher level planners such as the Directed Acyclic Graph Manager (DAGMan) [6]. This allows the users to be able to schedule both CPU resources and storage resources together. We made some enhancements to DAGMan, so that it can differentiate between computational jobs and data placement jobs. It can then submit computational jobs to a computational job scheduler, such as Condor [16] or Condor-G [12], and the data placement jobs to Stork. Figure 2 shows a sample DAG specification file with the enhancement of data placement nodes, and how this DAG is handled by DAGMan.

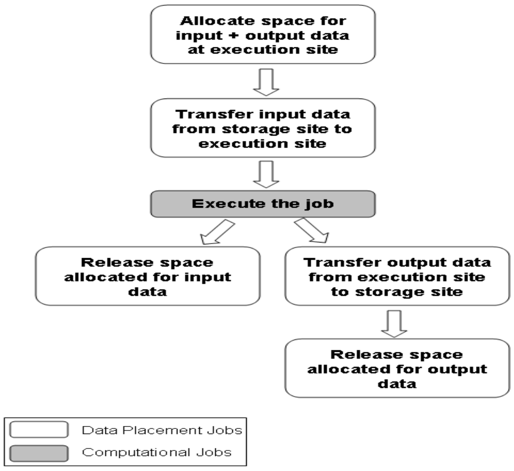


**Figure 1. Five Step Plan.** *Computation at a remote site with input and output data requirements can be achieved with a five step plan, which is represented as a six node DAG.*

In this way, it can be made sure that an input file required for a computation arrives to a storage device close to the execution site before actually that computation starts executing on that site. Similarly, the output files can be removed to a remote storage system as soon as the computation is completed. No storage device or CPU is occupied more than it is needed, and jobs do not wait idle for their input data to become available.

## 4.2. Interaction with Heterogeneous Resources

Stork is completely modular and can be extended easily. It is very straightforward to add support to Stork for your favorite storage system, data transport protocol, or middleware. This is a very crucial feature in a system designed to work in a heterogeneous Grid environment. The users or applications may not expect all storage systems to support the same interfaces to talk to each other. And we cannot expect all applications talking to all different kinds of storage systems, protocols, and middleware. There needs to be a negotiating system between them which can interact to those systems easily and even translate different protocols to each other. Stork has been developed to be capable of this. The

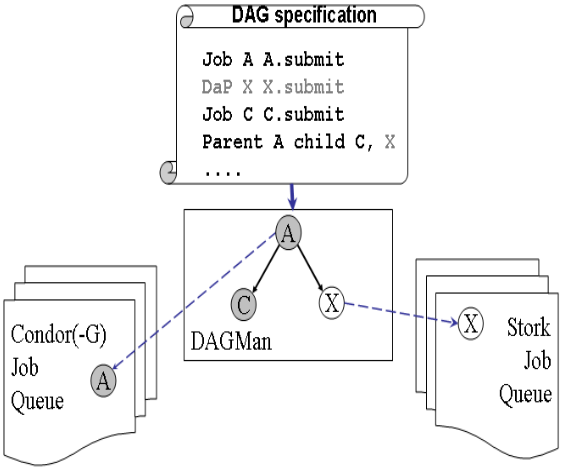


**Figure 2. Interaction with Higher Level Planners.** *In this prototype model, Stork interacts with a higher level planner: DAGMan. A DAG specification file consisting of both computational and data placement jobs is submitted to DAGMan. DAGMan then submits computational jobs to Condor/Condor-G, and data placement jobs to Stork.*

```
[
  dap_type     = "reserve";
  dest_host    = "db18.cs.wisc.edu";
  reserve_size = "100 MB";
  duration     = "2 hours";
  reserve_id   = 3;
]

[
  dap_type = "transfer";
  src_url  = "srb://ghidorac.sdsc.edu/home/kosart.condor/1.dat";
  dest_url = "nest://db18.cs.wisc.edu/1.dat";
]

[
  dap_type  = "release";
  dest_host = "db18.cs.wisc.edu";
  reserve_id = 3;
]
```

**Figure 3. Job representation in Stork.** *Three sample data placement (DaP) requests are shown: first one to allocate space, second one to transfer a file to the reserved space, and third one to de-allocate the reserved space.*

modularity of Stork allows users to insert a plug-in to support their favorite storage system, protocol, or middleware easily.

Stork already has support for several different storage systems, data transport protocols, and middleware. Users can use them immediately without any extra work. Stork can interact currently with data transfer protocols such as FTP [18], GridFTP, HTTP and DiskRouter [13]; data storage systems such as SRB, UniTree [4], and NeST; and data management middleware such as SRM.

Stork maintains a library of pluggable "data placement" modules. These modules get executed by data placement job requests coming to Stork. They can perform inter-protocol translations either using a memory buffer or third-party transfers whenever available. In order to transfer data between systems for which direct inter-protocol translation is not supported, two consecutive Stork jobs can be used instead. The first Stork job performs transfer from the source storage system to the local disk cache of Stork, and the second Stork job performs the transfer from the local disk cache of Stork to the destination storage system.

### 4.3. Flexible Job Representation and Multilevel Policy Support

Stork uses the ClassAd [20] job description language to represent the data placement jobs. The ClassAd language provides a very flexible and extensible data model that can be used to represent arbitrary services and constraints.

Figure 3 shows three sample data placement (DaP) requests. First request is to allocate 100 MB of disk space for 2 hours on a NeST server. Second request is to transfer a file from an SRB server to the reserved space on the NeST server. The third request is to de-allocate previously reserved space. In addition to the "reserve", "transfer", and "release", there are also other data placement job types such as "locate" to find where the data is actually located and "stage" to move the data from a tertiary storage to a secondary storage next to it in order to decrease data access time during actual transfers.

Stork enables users to specify job level policies as well as global ones. Global policies apply to all jobs scheduled by the same Stork server. Users can overwrite them by specifying job level policies in job description ClassAds. The example below shows how to overwrite global policies at the job level.

```
[
  dap_type = ``transfer'';
  ...
  ...
  max_retry  = 10;
  restart_in = ``2 hours'';
]
```

In this particular example, the user specifies that this particular job should be retried up to 10 times in case of failure, and if the transfer does not get completed in 2 hours, it should be killed and restarted.

### 4.4. Run-time Adaptation

Stork can decide which data transfer protocol to use for each corresponding transfer dynamically and automatically at the run-time. Before performing each transfer, Stork makes a quick check to identify which protocols are available for both the source and destination hosts involved in the transfer. Stork first checks its own host-protocol library to see whether all of the hosts involved the transfer are already in the library or not. If not, Stork tries to connect to those particular hosts using different data transfer protocols, to determine the availability of each specific protocol on that particular host. Then Stork creates the list of protocols available on each host, and stores these lists as a library:

```
[
  host_name = "quest2.ncsa.uiuc.edu";
  supported_protocols = "diskrouter, gridftp, ftp";
]
[
  host_name = "nostos.cs.wisc.edu";
  supported_protocols = "gridftp, ftp, http";
]
```

If the protocols specified in the source and destination URLs of the request fail to perform the transfer, Stork will start trying the protocols in its host-protocol library to carry out the transfer. The users also have the option not to specify any particular protocols in the request, letting Stork to decide which protocol to use at run-time:

```
[
  dap_type = "transfer";
  src_url  = "any://slic04.sdsc.edu/tmp/foo.dat";
  dest_url = "any://quest2.ncsa.uiuc.edu/tmp/foo.dat";
]
```

In the above example, Stork will select any of the available protocols on both source and destination hosts to perform the transfer. So, the users do not need to care about which hosts support which protocols. They just send a request to Stork to transfer a file from one host to another, and Stork will take care of deciding which protocol to use.

The users can also provide their preferred list of alternative protocols for any transfer. In this case, the protocols in this list will be used instead of the protocols in the host-protocol library of Stork:

```
[
  dap_type = "transfer";
  src_url  = "drouter://slic04.sdsc.edu/tmp/foo.dat";
  dest_url = "drouter://quest2.ncsa.uiuc.edu/tmp/foo.dat";
  alt_protocols = "nest-nest, gsiftp-gsiftp";
]
```

In this example, the user asks Stork to perform the a transfer from slic04.sdsc.edu to quest2.ncsa.uiuc.edu using the DiskRouter protocol primarily. The user also instructs Stork to use any of the NeST or GridFTP protocols in case the DiskRouter protocol does not work. Stork will try to perform the transfer using the DiskRouter protocol first. In case of a failure, it will switch to the alternative protocols and will try to complete the transfer successfully. If the primary protocol becomes available again, Stork will switch to it again. So, whichever protocol available will be used to successfully complete user's request.

### 4.5. Failure Recovery and Efficient Resource Utilization

Stork hides any kind of network, storage system, middleware, or software failures from user applications. It has a "retry" mechanism, which can retry any failing data placement job any given number of times before returning a failure. It also has a "kill and restart" mechanism, which allows users to specify a "maximum allowable run time" for their data placement jobs. When a job execution time exceeds this specified time, it will be killed by Stork automatically end restarted. This feature overcomes the bugs in some systems, which cause the transfers to hang forever and never return. This can be repeated any number of times, again specified by the user.

Stork can control the number of concurrent requests coming to any storage system it has access to, and makes sure that neither that storage system nor the network link to that storage system get overloaded. It can also perform space allocation and deallocations to make sure that the required storage space is available on the corresponding storage system. The space reservations are supported by Stork as long as the corresponding storage systems have support for it.

## 5. Case Studies

We will now show the applicability and contributions of Stork with two case studies. The first case study shows using Stork to create a data-pipeline between two heterogeneous storage systems. In this case, Stork is used to transfer data between two mass storage systems which do not have a common interface. This is done fully automatically and all failures during the course of the transfers are recovered without any human interaction. The second case study shows how Stork can be used for run-time adaptation of data transfers. If data transfer with one particular protocol fails, Stork uses other protocols available to successfully complete the transfer.

### 5.1. Building Data-pipelines

NCSA scientists wanted to transfer the Digital Palomar Sky Survey (DPOSS) [7] image data residing on SRB [2] mass storage system at SDSC in California to their UniTree mass storage system at NCSA in Illinois. The total data size was around 3 TB (2611 files of 1.1 GB each). Since

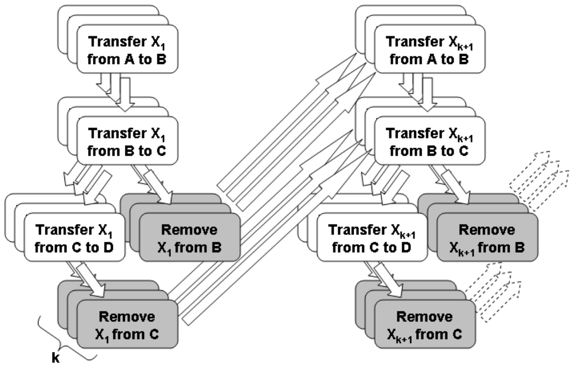


**Figure 4. Transfer in 5 Steps.** *Nodes representing the five steps of a single transfer are combined into a giant DAG to perform all transfers in the SRB - UniTree data-pipeline. k is the concurrency level.*

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 build a data pipeline. For this purpose, we have designed a data-pipeline using Stork.

In this pipeline, we set up two cache nodes between the source and destination storage systems. The first cache node (slic04.sdsc.edu) was at the SDSC site very close to the SRB server, and the second cache node (quest2.ncsa.uiuc.edu) was at the NCSA site near the UniTree server. This pipeline configuration allowed us to transfer data first 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 DiskRouter transfers, and finally from the NCSA cache node to the UniTree server using the underlying protocol of UniTree.

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 each file consisted of five basic steps, all of which we considered as real jobs to be submitted either to the Condor or Stork scheduling systems. All of these steps are represented as nodes in a DAG with arcs representing the dependencies between the steps. Then all of these five node DAGs were joined together to form a giant DAG as shown in Figure 4. The whole process was managed by DAGMan.

The SRB server, the UniTree server, and the SDSC cache node had gigabit ethernet(1000 Mb/s) interface cards installed on them. 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

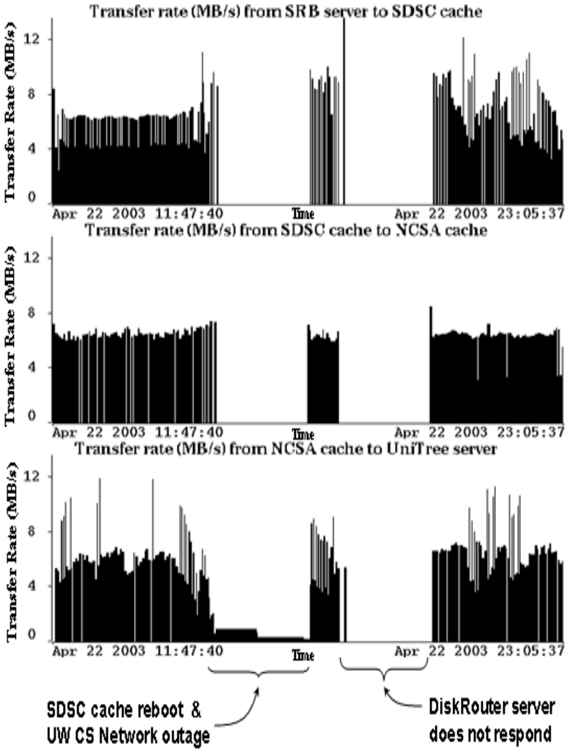


**Figure 5. Automated Failure Recovery in case of Network, Cache Node and Software Problems.** *The transfers recovered automatically again despite almost all possible failures occurring one after the other: UW CS network goes down, SDSC cache node goes down, and finally DiskRouter stops responding.*

NCSA cache node. We got an end-to-end transfer rate of 47.6 Mb/s from the SRB server to the UniTree server.

In this study, we have shown that we can successfully build a data-pipeline between two heterogeneous mass-storage systems, SRB and UniTree. Moreover, we have fully automated the operation of the pipeline and successfully transferred around 3 Terabytes of DPOSS data from the SRB server to the UniTree server without any human interaction.

During the transfers between SRB and UniTree, we had a wide variety of failures. At times either the source or destination mass-storage systems stopped accepting new transfers, due to either software failures or scheduled maintenance activity. We also had wide-area network outages, and software upgrades. Once in a while, a third-party DiskRouter transfer would hang. All of these failures were recovered automatically and the transfers were completed successfully without any human interaction.

Figure 5 shows multiple failures occurring during the

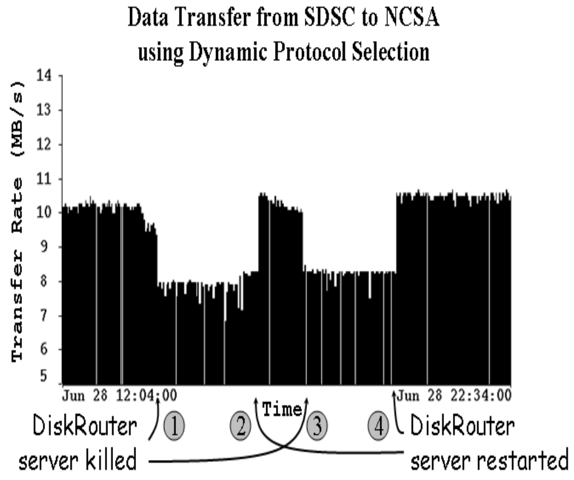


**Figure 6. Dynamic Protocol Selection.** *The DiskRouter server running on the SDSC machine gets killed twice at points (1) and (3), and it gets restarted at points (2) and (4). In both cases, Stork employed next available protocol (GridFTP in this case) to complete the transfers.*

course of the transfers. First the SDSC cache machine was rebooted and then there was a UW CS network outage which disconnected the management site and the execution sites for a couple of hours. The pipeline 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.

### 5.2. Run-time Adaptation of Data Transfers

We submitted 500 data transfer requests to the Stork server running at University of Wisconsin (skywalker.cs.wisc.edu). Each request consisted of transfer of a 1.1GB image file (total 550GB) from SDSC (slic04.sdsc.edu) to NCSA (quest2.ncsa.uiuc.edu) using the DiskRouter protocol. There was a DiskRouter server installed at Starlight (ncdm13.sl.startap.net) which was responsible for routing DiskRouter transfers. There were also GridFTP servers running on both SDSC and NCSA sites, which enabled us to use third-party GridFTP transfers whenever necessary.

At the beginning of the experiment, both DiskRouter and GridFTP services were available. Stork started transferring files from SDSC to NCSA using the DiskRouter protocol as directed by the user. After a while, we killed the DiskRouter server running at Starlight intentionally. Stork immediately switched the protocols and continued the transfers using GridFTP without any interruption. Switching to GridFTP caused a decrease in the performance of the transfers, as shown in Figure 6. The reasons of this decrease in performance is because of the fact that GridFTP does not perform auto-tuning whereas DiskRouter does. In this experiment, we set the number of parallel streams for GridFTP transfers to 10, but we did not perform any tuning of disk I/O block size or TCP buffer size. DiskRouter performs auto-tuning for the network parameters including the number of TCP-streams in order to fully utilize the available bandwidth. DiskRouter can also use sophisticated routing to achieve better performance.

After letting Stork use the alternative protocol (in this case GridFTP) to perform the transfers for a while, we restarted the DiskRouter server at the SDSC site. This time, Stork switched back to using DiskRouter for the transfers, since it was the preferred protocol of the user. Switching back to the faster protocol resulted in an increase in the performance. We repeated this a couple of more times, and observed that the system behaved in the same way every time.

This experiment shows that with alternate protocol fall-over capability, grid data-placement jobs can make use of the new high performance protocols while they work and switch to more robust lower performance protocol when the high performance one fails.

## 6. Future Work

We are planning to enhance the interaction between Stork and the higher level planners and computational schedulers more. This will result in co-scheduling of computational and data resources and will allow users to use both resources more efficiently.

Currently, the scheduling of data placement activities using Stork are performed at the file level. The users can move around only complete files. We are planning to add support for data level or block level scheduling. In this way, the users will be able to schedule movements of partial files, or even any specific blocks of a file.

We are planning to add more intelligence and adaptation to transfers. Different data transfer protocols may have different optimum concurrency levels for any two source and destination nodes. Stork will be able to decide the concurrency level of the transfers it is performing, taking into consideration the source and destination nodes of the transfer, the link it using, and more importantly, the protocol with which it is performing the transfers. In case of availability of multiple protocols to transfer data between different nodes, Stork will be able to choose the on with the best performance, or the most reliable one according to the user preferences.

Stork will be able to decide through which path, ideally the optimum one, to transfer data by an enhanced integration with the DiskRouter tool. It will be able to select nodes on which DiskRouters should be deployed, start DiskRouters on these nodes, and transfer the data through them by optimizing both the path and also the network utilization.

Another enhancement will be done with adding checkpointing support to data placement jobs. Whenever a transfer fails, it will not be started from scratch, but rather only the remaining parts of the file will be transfered.

## 7. Conclusion

We have introduced a specialized scheduler for data placement activities in Grid. Data placement efforts which has been done either manually or by using simple scripts are now regarded as first class citizens just like the computational jobs. They can be queued, scheduled, monitored and managed in a fault tolerant manner. We have showed the current challences with the data placement efforts in the Grid, and how Stork can provide solutions to them. We introduced a framework in which computational and data placement jobs are treated and scheduled differently by their corresponding schedulers, where the management and syncronization of both type of jobs is performed by higher level planners.

With two case studies, we have shown the applicability and contributions of Stork. Stork can be used to transfer data between heterogeneous systems fully automatically. It can recover from storage system, network and software failures without any human interaction. It can dynamically adapt data placement jobs to the environment at the execution time. We have shown that it generates better performance results by dynamically switching to alternative protocols in case of a failure.

## References


[1] B. Allcock, J. Bester, J. Bresnahan, A. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, and S. T. ke. Secure, efficient data transport and replica management for high-pe rformance 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] I. Bird, B. Hess, and A. Kowalski. Building the mass storage system at Jefferson Lab. In *Proceedings of 18th IEEE Symposium on Mass Storage Systems*, San Diego, California, April 2001.

[4] 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.

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

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

[7] 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.

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

[9] FNAL. Enstore mass storage system. http://www.fnal.gov/docs/products/enstore/.

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

[11] 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.

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

[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/overview.html, 2003.

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

[16] 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.

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

[18] J. Postel. FTP: File Transfer Protocol Specification. RFC-765, 1980.

[19] PPDG. PPDG Deliverables to CMS. http://www.ppdg.net/archives/ppdg/2001/doc00017.doc.

[20] R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed resource management for high throughput computing. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing (HPDC7)*, Chicago, Illinois, July 1998.

[21] SDSC. High Performance Storage System (HPSS). http://www.sdsc.edu/hpss/.

[22] A. Shishani, A. Sim, and J. Gu. Storage Resource Managers: Middleware Components for Grid Storage. In *Nineteenth IEEE Symposium on Mass Storage Systems*, 2002.

[23] D. Thain, , and M. Livny. The ethernet approach to grid computing. In *Proceedings of the Twelfth IEEE Symposium on High Performance Distributed Computing*, Seattle, Washington, June 2003.

[24] 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.