A Framework for Reliable and Efficient Data Placement in Distributed Computing Systems

Tevfik Kosar and Miron Livny
Computer Sciences Department, University of Wisconsin-Madison
1210 West Dayton Street, Madison WI 53706
Tel:+1(608)262-1204, Fax:+1(608)262-9777
{kosart, miron}@cs.wisc.edu

Abstract

Data placement is an essential part of today's distributed applications since moving the data close to the application has many benefits. The increasing data requirements of both scientific and commercial applications, and collaborative access to these data make it even more important. In the current approach, data placement is regarded as a side affect of computation. Our goal is to make data placement a first class citizen in distributed computing systems just like the computational jobs. They will be queued, scheduled, monitored, managed, and even checkpointed. Since data placement jobs have different characteristics than computational jobs, they cannot be treated in the exact same way as computational jobs. For this purpose, we are proposing a framework which can be considered as a "data placement subsystem" for distributed computing systems, similar to the I/O subsystem in operating systems. This framework includes a specialized scheduler for data placement, a high level planner aware of data placement jobs, a resource broker/policy enforcer and some optimization tools. Our system can perform reliable and efficient data placement, it can recover from all kinds of failures without any human intervention, and it can dynamically adapt to the environment at the execution time.

Key words. *Distributed computing, reliable and efficient data placement, scheduling, run-time adaptation, protocol auto-tuning, data intensive applications, I/O subsystem.*

1. Introduction

The data requirements of both scientific and commercial applications have been increasing drastically in recent years. 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, and in less than a decade they are expected to reach the exabyte (1 million terabytes) scale [37].

The problem is not only the massive I/O needs of the data intensive applications, but also the number of users who will access and share the same datasets. For a range of applications from genomics to biomedical, and from metallurgy to cosmology, 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. Therefore, there is a big necessity to move large amounts of data around wide area networks for processing and for replication, which brings with it the problem of reliable and efficient data placement. Data needs to be located, moved to the application, staged and replicated; 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 computation and network resources need to be carefully scheduled and managed, the scheduling of data placement activities all across the distributed computing systems is crucial, since the access to data is generally 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.

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. They cannot adapt to a dynamically changing distributed computing environment. They do not have a single point of control, and generally require baby-sitting throughout the process. There are even cases where people found a solution for data placement by dumping data to tapes and sending them via postal services [18].

Data placement activities must be first class citizens in the distributed computing environments just like the com-

putational jobs. They need to be queued, scheduled, monitored, and even check-pointed. More importantly, it must be made sure that they complete successfully and without any need for human intervention. Currently, data placement is generally not considered part of the end-to-end performance, and requires lots of baby-sitting. In our approach, the end-to-end processing of the data will be completely automated, so that the user can just launch a batch of computational/data placement jobs and then forget about it.

Moreover, data placement jobs should be treated differently from computational jobs, since they have different semantics and different characteristics. Data placement jobs and computational jobs should be differentiated from each other and each should be submitted to specialized schedulers that understand their semantics. For example, if the transfer of a large file fails, we may not simply want to restart the job and re-transfer the whole file. Rather, we may prefer transferring 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. We may want to dynamically tune up network parameters or decide concurrency level for specific source, destination and protocol triples. A traditional computational job scheduler does not handle these cases. For this purpose, we have developed a "data placement subsystem" for distributed computing systems, similar to the I/O subsystem in operating systems. This subsystem includes a specialized scheduler for data placement, a higher level planner aware of data placement jobs, a resource broker/policy enforcer and some optimization tools.

This data placement subsystem provides complete reliability, a level of abstraction between errors and users/applications, ability to achieve load balancing on the storage servers, and to control the traffic on network links. It also allows the users to perform these two types of jobs asynchronously.

We show applicability and contributions of our work with three important case studies. Two of these case studies are from real world: a data processing pipeline for educational video processing and a data transfer pipeline for astronomy image processing. Our system 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.

2. Background

I/O has been very important throughout the history of computing, and special attention given to it to make it more reliable and efficient both in hardware and software levels.

In the old days, the CPU was responsible for carrying out all data transfers between I/O devices and the main memory at the hardware level. The overhead of initiating, monitoring and actually moving all data between the device and the main memory was too high to permit efficient utilization of CPU. To alleviate this problem, additional hardware was provided in each device controller to permit data to be directly transferred between the device and main memory, which led to the concepts of DMA (Direct Memory Access) and I/O processors (channels). All of the I/O related tasks are delegated to the specialized I/O processors, which were responsible for managing several I/O devices at the same time and supervising the data transfers between each of these devices and main memory [7].

On the operating systems level, initially the users had to write all of the code necessary to access complicated I/O devices. Later, low level I/O coding needed to implement basic functions was consolidated to an I/O Control System (IOCS). This greatly simplified users' jobs and sped up the coding process [14]. Afterwards, an I/O scheduler was developed on top of IOCS that was responsible for execution of the policy algorithms to allocate channel (I/O processor), control unit and device access patterns in order to serve I/O requests from jobs efficiently [31].

When we consider scheduling of I/O requests at the distributed systems level, we do not see the same recognition given them. They are not even considered as tasks that need to be scheduled and monitored independently. There has been a lot of work on remote access to data [22] [33], but this approach does not scale well as the target data set size increases. Moving the application closer to the data is not always practical, since storage systems generally do not have sufficient computational resources nearby. The general approach for large data sets is to move data to the application. Therefore, data need to be located and sent to processing sites; the results should be shared with other sites; storage space should be allocated and de-allocated whenever necessary; and everything should be cleaned up afterwards. Although these data placement activities are of great importance for the end-to-end performance of an application, they are generally considered as side effects of computation. They are either embedded into computational jobs, or performed using simple scripts.

Our framework and the system we have developed give data placement its recognition in distributed systems. This system can be considered as an I/O (or data placement) subsystem in a distributed computing environment.

3. Data Placement Subsystem

Most of the data intensive applications in distributed computing systems 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

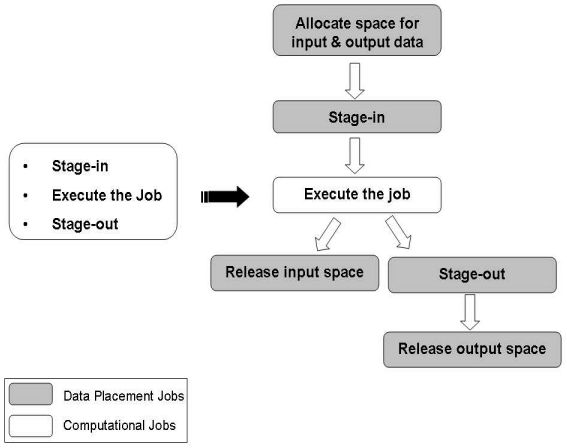

Figure 1. Separating data placement form computation. *Computation at a remote site with input and output data requirements can be achieved with a new five-step plan, in which computation and data placement are separated. This is represented as a six node DAG in the figure.*

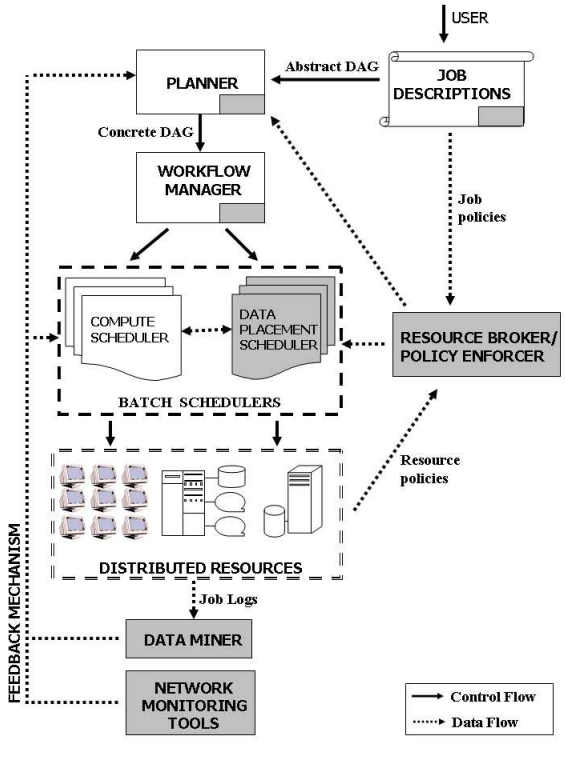

Figure 2. Components of the Data Placement Subsystem. *The components of our data placement subsystem are shown in gray color in the figure.*

the same or another remote site. If the application does not want to take any risk of running out of disk space at the execution site, it should also 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 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.

In our framework, the data placement jobs are represented in a different way than computational jobs in the job specification language, so that the high level planners (i.e. Pegasus [13], Chimera [23]) can differentiate these two classes of jobs. The high level planners create concrete DAGs with also data placement nodes in them. Then, the planner submits this concrete DAG to a workflow manager (i.e. DAGMan [44]). The workflow manager submits computational jobs to a compute job queue, and the data placement jobs to a data placement job queue. Jobs in each queue are scheduled by the corresponding scheduler. Since our focus in this work is on the data placement part, we do not get into details of the computational job scheduling.

The data placement scheduler acts both as a I/O control system and I/O scheduler in a distributed computing environment. Each protocol and data storage system have different user interface, different libraries and different API. In the current approach, the users need to deal with all complexities of linking to different libraries, and using different interfaces of data transfer protocols and storage servers. Our data placement scheduler provides a uniform interface for all different protocols and storage servers, and puts a level of abstraction between the user and them.

The data placement scheduler schedules the jobs in its queue according to the information it gets from the workflow manager and from the resource broker/policy enforcer. The resource broker matches resources to jobs, and helps in locating the data and making decisions such as where to move the data. It consults a replica location service (i.e. RLS [11]) whenever necessary. The policy enforcer helps in applying the resource specific or job specific policies, such as how many concurrent connections are allowed to a specific storage server.

The log files of the jobs are collected by the data miner. The data miner parses these logs and extracts useful information from them such as different events, timestamps, error messages and utilization statistics. Then this information is entered into a database. The data miner runs a set of queries on the database to interpret them and then feeds the results back to the scheduler and the resource broker/policy enforcer.

The network monitoring tools collect statistics on maxi-

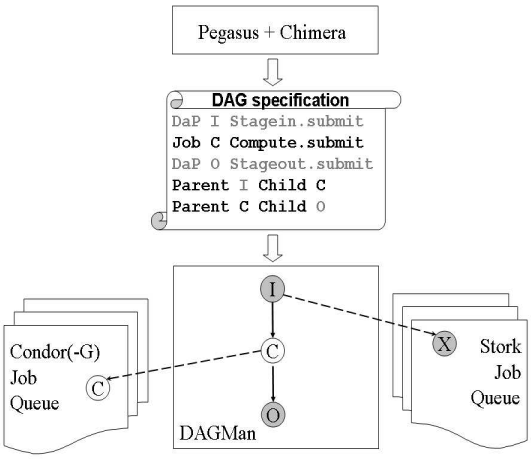

Figure 3. Interaction with Higher Level Planners. *Our data placement scheduler (Stork) can interact with a higher level planners and workflow managers. A concrete DAG created by Chimera and Pegasus is sent to DAGMan. This DAG consists of both computational and data placement jobs. DAGMan submits computational jobs to a computational batch scheduler (Condor/Condor-G), and data placement jobs to Stork.*

mum available end-to-end bandwidth, actual bandwidth utilization, latency and number of hops to be traveled by utilizing tools such as Pathrate [16] and Iperf [34]. Again, the collected statistics are fed back to the scheduler and the resource broker/policy enforcer.

The components of our data placement subsystem and their interaction with other components are shown in Figure 2. The most important component of this system is the data placement scheduler, which can understand the characteristics of the data placement jobs and can make smart scheduling decisions accordingly. In the next section, we present the features of this scheduler in detail.

4. Data Placement Scheduler (Stork)

We have implemented a prototype of the data placement scheduler we are proposing. We call this scheduler Stork. Stork provides solutions for many of the data placement problems encountered in the distributed computing environments.

Interaction with Higher Level Planners. Stork can interact with higher level planners and workflow managers. 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 [29] or Condor-G [24], and the data placement jobs to Stork. Figure 3 shows a sample DAG specification file with the enhancement of data placement nodes, and how this DAG is handled by DAGMan.

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.

Interaction with Heterogeneous Resources. Stork acts like an I/O control system (IOCS) between the user applications and the underlying protocols and data storage servers. It provides complete modularity and extendibility. The users can add support for their favorite storage system, data transport protocol, or middleware very easily. This is a very crucial feature in a system designed to work in a heterogeneous distributed 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 to talk to all the different storage systems, protocols, and middleware. There needs to be a negotiating system between them which can interact with those systems easily and even translate different protocols to each other. Stork has been developed to be capable of this. The modularity of Stork allows users to insert a plug-in to support any 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 [36], GridFTP [2], HTTP and DiskRouter [25]; data storage systems such as SRB [3], UniTree [9], and NeST [12]; and data management middleware such as SRM [41].

Stork maintains a library of pluggable "data placement" modules. These modules get executed by data placement job requests coming into Stork. They can perform inter-protocol translations either using a memory buffer or third-party transfers whenever available. Inter-protocol translations are not supported between all systems or protocols yet. Figure 4 shows the available direct inter-protocol translations that can be performed using a single Stork job.

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

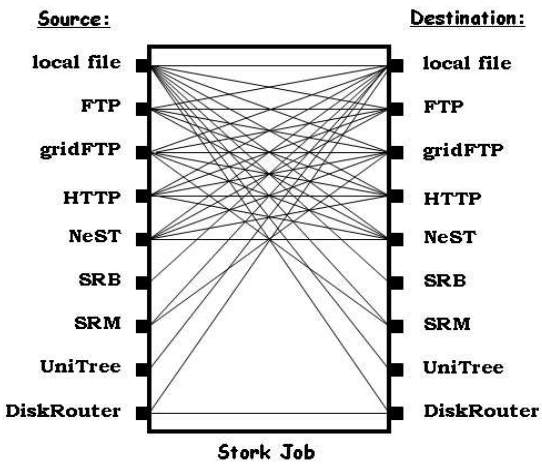

Figure 4. Protocol Translation using Stork Memory Buffer or Third-party Transfers. *Transfers between some storage systems and protocols can be performed directly using one Stork job via memory buffer or third-party transfers.*

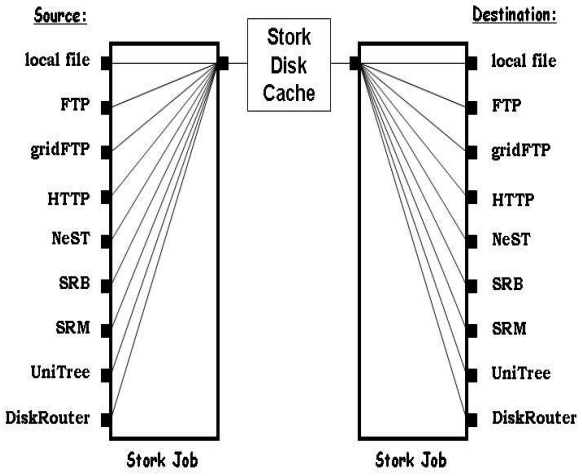

Figure 5. Protocol Translation using Stork Disk Cache. *Transfers between all storage systems and protocols supported can be performed using two Stork jobs via an intermediate disk cache.*

storage system. This is shown in Figure 5.

Flexible Job Representation and Multilevel Policy Support. Stork uses the ClassAd [38] 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 6 shows three sample data placement (DaP) requests. The first request is to allocate 100 MB of disk space for 2 hours on a NeST server. The 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 the 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 override them by specifying job level policies in job description ClassAds. The example below shows how to override global policies at the job level.

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

In this example, the user specifies that the 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.

Dynamic Protocol Selection. Stork can decide which data transfer protocol to use for each 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 in 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     = "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 6. 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.*

```
  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. Therefore, 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. Hence, whichever protocol is available will be used to successfully complete user's request.

Run-time Protocol Auto-tuning. Statistics for each link involved in the transfers are collected regularly and written into a file, creating a library of network links, protocols and auto-tuning parameters.

```
[
  link = "slic04.sdsc.edu - quest2.ncsa.uiuc.edu";
  protocol = "gsiftp";

  bs       = 1024KB;     //block size
  tcp_bs   = 1024KB;    //TCP buffer size
  p        = 4;         //parallelism
]
```

Before performing every transfer, Stork checks its auto-tuning library to see if there are any entries for the particular hosts involved in this transfer. If there is an entry for the link to be used in this transfer, Stork uses these optimized parameters for the transfer. Stork can also be configured to collect performance data before every transfer, but this is not recommended due to the overhead it would bring to the system.

Failure Recovery. Stork hides any kind of temporary 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.

Efficient Resource Utilization. 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 our data placement subsystem with three 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 our data placement subsystem can be used for run-time adaptation of data transfers. If data

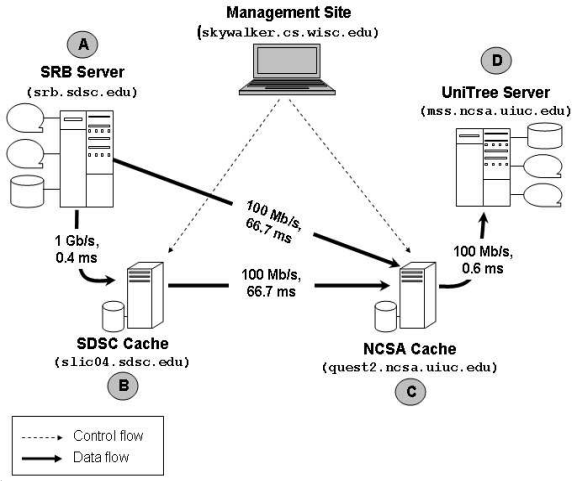

Figure 7. Data-pipeline with Two Intermediate Nodes. *Building a data-pipeline with two intermediate nodes, one close to the source and one close to the destination, may provide additional functionality and increase performance.*

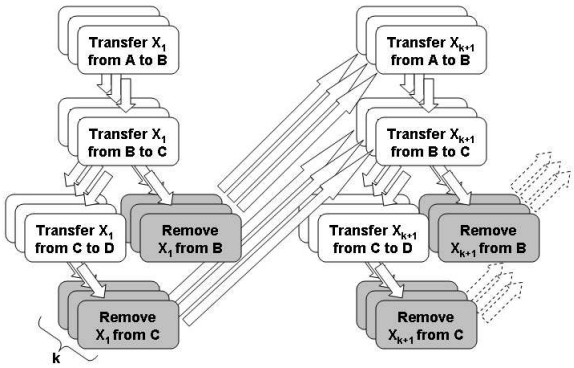

Figure 8. Transfer in five Steps. *Nodes representing the five steps of a single transfer are combined into a giant DAG to perform all transfers in the SRB - UniTree data-pipeline. k is the concurrency level.*

transfer with one particular protocol fails, Stork uses other protocols available to successfully complete the transfer. In case where the network parameters are not well tuned, our data placement subsystem can perform auto-tuning during run-time. In the third case study, we have built a fully automated data processing pipeline to process terabytes of educational video.

5.1. Building Data-pipelines

NCSA scientists wanted to transfer the Digital Palomar Sky Survey (DPOSS) [15] image data residing on SRB [3] 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 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. This pipeline configuration is shown in Figure 7.

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 8. 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 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 the destination mass-storage systems stopped accepting new transfers, due to either software failures or scheduled maintenance activity. We also had wide-area network out-

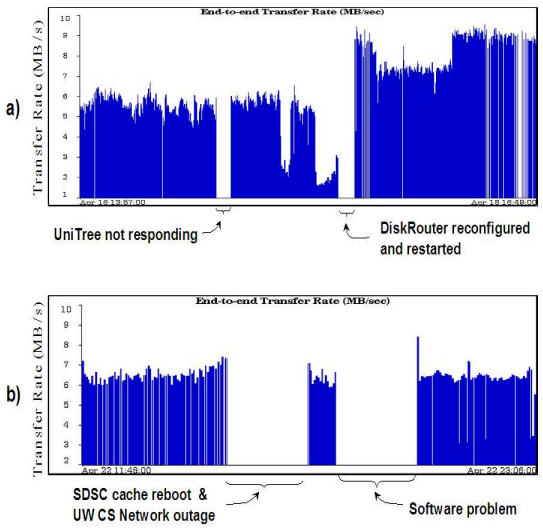

Figure 9. Automated Failure Recovery. *The transfers recovered automatically despite almost all possible failures occurring one after the other: a) The UniTree server stops responding, the DiskRouter server gets reconfigured and restarted during the transfers b) UW CS network goes down, SDSC cache node goes down, and finally there was a problem with the DiskRouter server.*

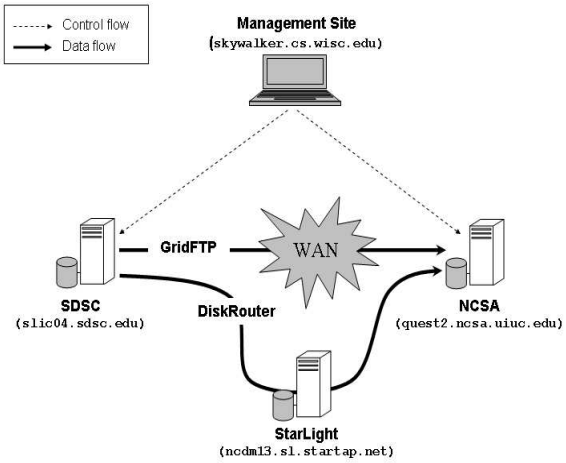

Figure 10. Experiment Setup. *DiskRouter and GridFTP protocols are used to transfer data from SDSC to NCSA. Stork was running at the Management site, and making scheduling decisions for the transfers.*

ages, and software upgrades. Occasionally, 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 9 shows multiple failures occurring during the 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 have performed two different experiments to evaluate the effectiveness of our dynamic protocol selection and run-time protocol tuning mechanisms. We also collected performance data to show the contribution of these mechanisms to wide area data transfers.

Dynamic Protocol Selection. 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. The experiment setup is shown in Figure 10.

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 11. The reason of this decrease in performance is 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

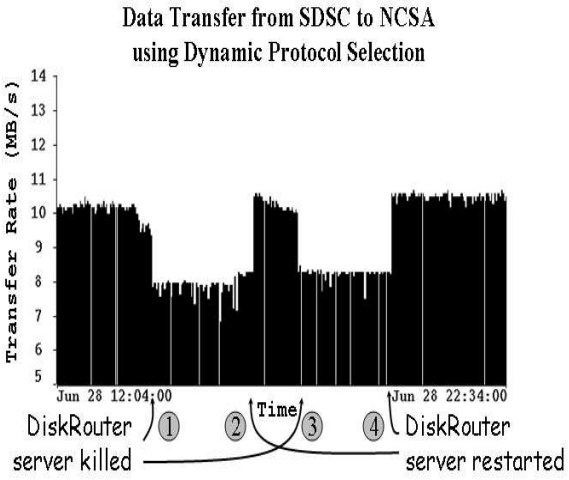

Figure 11. Dynamic Protocol Selection. *The DiskRouter server running on the SDSC machine is killed twice at points (1) and (3), and it is restarted at points (2) and (4). In both cases, Stork employed next available protocol (GridFTP in this case) to complete the transfers.*

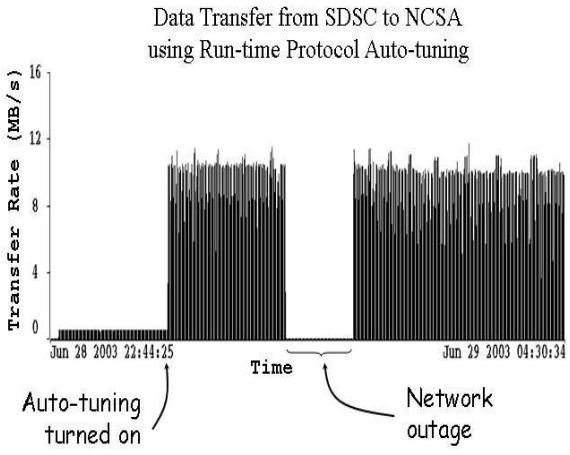

Figure 12. Run-time Protocol Auto-tuning. *Stork starts the transfers using the GridFTP protocol with auto-tuning turned off intentionally. Then we turn th e auto-tuning on, and the performance increases drastically*

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 couple of more times, and observed that the system behaved in the same way every time.

This experiment shows that with alternate protocol fallover 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.

Run-time Protocol Auto-tuning. In the second experiment, we submitted another 500 data transfer requests to the Stork server. Each request was to transfer a 1.1GB image file (total 550 GB) using GridFTP as the primary protocol. We used third-party globus-url-copy transfers without any tuning and 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 underlying monitoring and tuning infrastructure. Then it applied these values to the subsequent transfers. Figure 12 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.

5.3. Educational Video Processing Pipeline

Wisconsin Center for Educational Research (WCER) wanted to process nearly 500 terabytes of educational video. They wanted to get mpeg1, mpeg2 and mpeg4 encodings from the original DV format and make all formats electronically available for collaborating researchers. They wanted the videos to be stored both in their storage server and at the SRB mass storage at San Diego supercomputing center (SDSC). They did not have the necessary computational power nor the infrastructure to process and replicate these videos.

We have created a video processing pipeline for them using Condor and Stork technologies. The video files (each 13GB) first get transferred from WCER to a staging area at UW CS Department. Since the internal file transfer mechanism of Condor did not support files larger than 2 GB, we

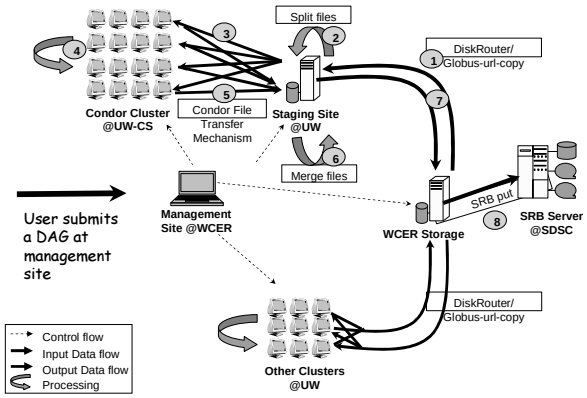

Figure 13. WCER Configuration

split the files here into 2 GB junks and transferred them to individual compute nodes. At each compute node, these 2GB junks got merged again and processed. The encoded mpeg files got transferred back to the staging node, and from here back to WCER. A copy of each file got replicated to the SRB server at SDSC. The whole process was managed by DAGMan, and it was performed fully automatically without any human intervention. This pipeline is shown in Figure 13.

6. 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 [18].

The Reliable File Transfer Service(RFT) [30] 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) [26] can replicate data sets to the member sites of a Virtual Organization or DataGrid. It was primarily developed for replicating LIGO [28] data, and it makes use of Globus [21] 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) [41] on top of them. Currently, a couple of data storage systems, such as HPSS [39], Jasmin [8] and Enstore [20], support SRMs on top of them. SRMs can also manage distributed caches using "pinning of files".

The SDSC Storage Resource Broker (SRB) [3] 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 [42] to distributed 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.

Network Weather Service (NWS) [46] is a distributed system which periodically gathers readings from network and CPU resources, and uses numerical models to generate forecasts for a given time frame. Vazhkudai [45] found that the network throughput predicted by NWS was much less than the actual throughput achieved by GridFTP.

Semke [40] introduces automatic TCP buffer tuning. Here the receiver is expected to advertise large enough windows. Fisk [19] points out the problems associated with [40] and introduces dynamic right sizing which changes the receiver window advertisement according to estimated sender congestion window.

Fearman et. al [17] introduce the Adaptive Regression Modeling (ARM) technique to forecast data transfer times for network-bound distributed data-intensive applications. Ogura et. al [35] try to achieve optimal bandwidth even when the network is under heavy contention, by dynamically adjusting transfer parameters between two clusters, such as the number of socket stripes and the number of network nodes involved in transfer.

In [10], Carter et. al. introduce tools to estimate the maximum possible bandwidth along a given path, and to calculate the current congestion along a path. Using these tools, they demonstrate how dynamic server selection can be performed to achieve application-level congestion avoidance.

Application Level Schedulers (AppLeS) [6] have been developed to achieve efficient scheduling by taking into account both application-specific and dynamic system information. AppLeS agents use dynamic system information provided by the NWS.

Beck et. al. introduce Logistical Networking [4] which performs global scheduling and optimization of data movement, storage and computation based on a model that takes into account all the network's underlying physical resources. Kangaroo [43] tries to achieve high throughput by making opportunistic use of disk and network resources.

GFarm [32] provides a global parallel filesystem with

online petascale storage. Their model specifically targets applications where data primarily consists of a set of records or objects which are analyzed independently. Gfarm takes advantage of this access locality to achieve a scalable I/O bandwidth using a parallel filesystem integrated with process scheduling and file distribution.

OceanStore [27] aims to build a global persistent data store that can scale to billions of users. The basic idea is that any server may create a local replica of any data object. These local replicas provide faster access and robustness to network partitions. Both Gfarm and OceanStore require creating several replicas of the same data, but still they do not address the problem of scheduling the data movement when there is no replica close to the computation site.

BAD-FS [5] builds a batch aware distributed filesystem for data intensive workloads. This is general purpose and serves workloads more data intensive than conventional ones. For performance reasons it prefer to access source data from local disk rather than over a network filesystem. Further, BAD-FS at present does not schedule wide-area data movement which we feel is necessary for large data sets.

7. Future Work

We are planning to enhance the interaction between our data placement scheduler and the higher level planners and workflow managers like DAGMan, Pagasus and Chimera. This will result in better co-scheduling of computational and data resources and will allow users to use both resources more efficiently.

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. Our data placement subsystem 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, the data placement scheduler will be able to choose the on with the best performance, or the most reliable one according to the user preferences.

Our data placement subsystem will be able to decide through which path, ideally the optimum one, to transfer data by an enhanced integration with the underlying network and data transfer tools. Another enhancement will be done with adding check-pointing 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.

8. Conclusion

We have introduced a data placement subsystem for reliable and efficient data placement in distributed computing systems. 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 how our system can provide solutions to the data placement problems of the distributed systems community. We introduced a framework in which computational and data placement jobs are treated and scheduled differently by their corresponding schedulers, where the management and synchronization of both type of jobs is performed by higher level planners.

With several case studies, we have shown the applicability and contributions of our data placement subsystem. It 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. It can help auto-tuning some network parameters to achieve higher data transfer rate. We have also shown that how our system can be used in interaction with other schedulers and higher level planners to create reliable, efficient and fully automated data processing pipelines.

9 Acknowledgments

We would like to thank Robert Brunner from NCSA, Philip Papadopoulos from SDSC, and Chris Torn from WCER for collaborating with us, letting us use their resources and making it possible to try our data placement subsystem on real-life grid applications. We also would like to thank George Kola for helping us in setting up and using the DiskRouter tool.

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, effi cient data transport and replica management for high-pe rformance data-intensive computing. In *IEEE Mass Storage Conference*, San Diego, CA, April 2001.

[2] W. Allcock, I. Foster, and R. Madduri. Reliable data transport: A critical service for the grid. In *Building Service Based Grids Workshop, Global Grid Forum 11*, June 2004.

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

[4] M. Beck, T. Moore, J. Plank, and M. Swany. Logistical networking. In *Active Middleware Services*, S. Hariri and C. Lee and C. Raghavendra, editors. Kluwer Academic Publishers., 2000.

[5] J. Bent, D. Thain, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Explicit control in a batch-aware distributed file system. In *Proceedings of the First USENIX/ACM Conference on Networked Systems Design and Implementation*, March 2004.

[6] F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao. Application level scheduling on distributed heterogeneous networks. In *Proceedings of Supercomputing'96*, Pittsburgh, Pennsylvenia, November 1996.

[7] L. Bic and A. C. Shaw. The Organization of Computer Systems. In *The Logical Design of Operationg Systems.*, Prentice Hall., 1974.

[8] 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.

[9] 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.

[10] R. L. Carter and M. E. Crovella. Dynamic server selection using bandwidth probing in wide-area networks. Technical Report TR-96-007, Computer Science Department, Boston University, 1996.

[11] L. Chervenak, N. Palavalli, S. Bharathi, C. Kesselman, and R. Schwartzkopf. Performance and scalability of a Replica Location Service. In *Proceedings of the International Symposium on High Performance Distributed Computing Conference (HPDC-13)*, Honolulu, Hawaii, June 2004.

[12] Condor. NeST: Network Storage. http://www.cs.wisc.edu/condor/nest/, 2003.

[13] E. Deelman, J. Blythe, Y. Gil, and C. Kesselman. Pegasus: Planning for execution in grids. In *GriPhyN technical report*, 2002.

[14] H. M. Deitel. I/O Control System. In *An Introduction to Operating Systems.*, Addison-Wesley Longman Publishing Co., Inc., 1990.

[15] 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.

[16] C. Dovrolis, P. Ramanathan, and D. Moore. What do packet dispersion techniques measure? In *INFOCOMM*, 2001.

[17] M. Faerman, A. Su, R. Wolski, and F. Berman. Adaptive performance prediction for distributed data-intensive applications. In *Proceedings of the IEE/ACM Conference on High Performance Networking and Computing*, Portland, Oregon, November 1999.

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

[19] M. Fisk and W. Weng. Dynamic right-sizing in TCP. In *ICCCN*, 2001.

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

[21] 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.

[22] I. Foster, D. Kohr, R. Krishnaiyer, and J.Mogill. Remote I/O: Fast access to distant storage. In *Proc. IOPADS'97*, pages 14–25. ACM Press, 1997.

[23] I. Foster, J. Vockler, M. Wilde, and Y. Zhao. Chimera: A virtual data system for representing, querying, and automating data derivation. In *14th International Conference on Scientific and Statistical Database Management (SSDBM 2002)*, Edinburgh, Scotland, July 2002.

[24] 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.

[25] 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.

[26] S. Koranda and B. Moe. Lightweight Data Replicator. http://www.lsc-group.phys.uwm.edu/lscdatagrid/LDR/overview.html, 2003.

[27] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, November 2000.

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

[29] 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.

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

[31] S. E. Madnick and J. J. Donovan. I/O Scheduler. In *Operating Systems.*, McGraw-Hill, Inc., 1974.

[32] Y. Morita, H. Sato, Y. Watase, O. Tatebe, S. Sekiguchi, S. Matsuoka, N. Soda, and A. Dell'Acqua. Building a high performance parallel file system using grid datafarm and root i/o. In *Proceedings of the 2003 Computing in High Energy and Nuclear Physics (CHEP03)*, La Jolla, CA, March 2003.

[33] J. Nieplocha, I. Foster, and H. Dachsel. Distant I/O: One-sided access to secondary storage on remote processors. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*, pages 148–154, July 1998.

[34] NLANR/DAST. Iperf: The TCP/UDP bandwidth measurement tool. http://dast.nlanr.net/Projects/Iperf/, 2003.

[35] S. Ogura, H. Nakada, and S. Matsuoka. Evaluation of the inter-cluster data transfer on Grid environment. In *Proceedings of the Third IEEE/ACM Symposium on Cluster Computing and the Grid (CCGrid2003)*, Tokyo, Japan, May 2003.

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

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

[38] 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.

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

[40] J. Semke, J. Mahdavi, and M. Mathis. Automatic TCP buffer tuning. In *SIGCOMM*, pages 315–323, 1998.

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

[42] 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.

[43] 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.

[44] 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.

[45] S. Vazhkudai, J. Schopf, and I. Foster. Predicting the Performance of Wide Area Data Transfers. In *Proceedings of the 16th Int'l Parallel and Distributed Processing Symposium (IPDPS 2002)*, 2002.

[46] R. Wolski. Dynamically forecasting network performance to support dynamic scheduling using the Network Weather Service. In *Proceedings of the Sixth IEEE Symposium on High Performance Distributed Computing (HPDC6)*, Portland, Oregon, August 1997.