DATA PLACEMENT IN WIDELY DISTRIBUTED SYSTEMS

by
Tevfik Kosar

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy
(Computer Sciences)

at the
UNIVERSITY OF WISCONSIN-MADISON
2005

Abstract

The unbounded increase in the computation and data requirements of scientific applications has necessitated the use of widely distributed compute and storage resources to meet the demand. In such an environment, data is no more locally accessible and has thus to be remotely retrieved and stored. Efficient and reliable access to data sources and archiving destinations in a widely distributed environment brings new challenges. Placing data on temporary local storage devices offers many advantages, but such "data placements" also require careful management of storage resources and data movement, i.e. allocating storage space, staging-in of input data, staging-out of generated data, and de-allocation of local storage after the data is safely stored at the destination.

Existing systems closely couple data placement and computation, and consider data placement as a side effect of computation. Data placement is either embedded in the computation and causes the computation to delay, or performed as simple scripts which do not have the privileges of a job. In this dissertation, we propose a framework that de-couples computation and data placement, allows asynchronous execution of each, and treats data placement as a full-fledged job that can be queued, scheduled, monitored and check-pointed like computational jobs. We regard data placement as an important part of the end-to-end process, and express this in a workflow language.

As data placement jobs have different semantics and different characteristics than computational jobs, not all traditional techniques applied to computational jobs apply to data placement jobs. We analyze different scheduling strategies for data placement, and introduce a batch scheduler specialized for data placement. This scheduler implements techniques specific to queuing, scheduling, and optimization of data placement jobs, and provides a level of abstraction between the user applications and the underlying data transfer and storage resources.

We provide a complete data placement subsystem for distributed computing systems, similar to I/O subsystem in operating systems. This system offers transparent failure handling, reliable, efficient scheduling of data resources, load balancing on the storage servers, and traffic control on network links. It provides policy support, improves fault-tolerance and enables higher-level optimizations including maximizing the application throughput. Through deployment in several real-life applications such as US-CMS, DPOSS Astronomy Pipeline, and WCER Educational Video Pipeline, our approach has proved to be effective, providing a promising new research direction.

Acknowledgments

It is a great pleasure for me to thank the people who helped me during different stages of my academic career and made this dissertation possible.

Firstly, I would like to thank my advisor, Dr. Miron Livny, for giving me the chance to work under his supervision, for providing invaluable insights, encouragement, and guidance throughout my graduate study, and for the great opportunities and unbounded resources he has provided me to pursue my research.

I am grateful to Dr. Remzi Arpaci-Dusseau, Dr. Jeffrey Naughton, Dr. Sridhara Dasu, and Dr. Andrea Arpaci-Dusseau for their evaluation of my work, and for the comments and feedback they have provided on my dissertation. They helped me think in the ways I have never thought before, and helped me to improve the quality of my work.

I thank Dr. Rong Chang and Dr. Liana Fong from IBM T.J. Watson Research Center for providing me with the research opportunities as an intern at one of the best research centers in the country.

Dr. Robert Brunner from University of Illinois at Urbana-Champaign, Dr. Philip Papadopoulos from San Diego Supercomputing Center, and Dr. Chris Thorn from Wisconsin Center for Educational Research opened their resources to me, which enabled the empirical verification of my research.

George Kola has been a great colleague for me on many joint projects and papers. I have always enjoyed the discussions and brain-storming with him. Some parts of this dissertation are the fruits of the joint work between me and George. I thank George for his acquaintance.

I thank all of the members of the Condor team for their perfect fellowship, and the very welcoming work and research atmosphere they have created. I also would like to thank all of my friends both inside and outside of the department, who made Madison for me a more livable place.

My parents, Mustafa and Meliha Kosar, have always been a great support for me. They have provided continuous encouragement and showed endless patience when it was most required. My brother Murat Kosar has always been more than a brother for me: a friend, a role model, and a mentor. I do not know any words which can express my thanks and gratefulness to my parents and to my brother.

Nilufer Beyza Kosar, my daughter, has been the joy of my life for the last two years. Even during the most stressful moments, she was be able to make me laugh, and forget everything else.

And finally, I thank my wonderful wife, Burcu Kosar, for always being with me, and never complaining. Without her love and support, nothing would be possible.

Contents

List of Figures

Chapter 1

Introduction

The computational and data requirements of scientific applications have been increasing drastically over the last decade. Just five years ago, the total amount of data to be processed by scientific applications was on the order of a couple hundred terabytes per year. This amount is expected to reach the order of several million terabytes per year in less than ten years. This exponential increase in the size of scientific data has already outpaced the increase in the computational power and the storage space predicted by the Moore's Law [56, 86].

Figure 1.1 shows the increase solely in the genomics datasets over the last decade [11] and its comparison with the expected growth according to the Moore's Law. When we add the data from other fields of science such as astronomy, high energy physics, chemistry, earth sciences, and educational technology to this picture, the total amount of data to be processed is hard to estimate and far beyond imagination. Table 1 shows the data requirements of some of the scientific applications in different areas.

Large amounts of scientific data also require large amounts of computational power in order to be processed in a reasonable time scale. The number of CPUs necessary for ATLAS [16] and CMS [3] applications alone is on the order of hundred thousands. These high computational and data requirements of scientific applications necessitated the use of widely distributed resources owned by collaborating parties to meet the demand. There has been considerable amount of work done on distributed computing [35, 53, 78, 109], batch

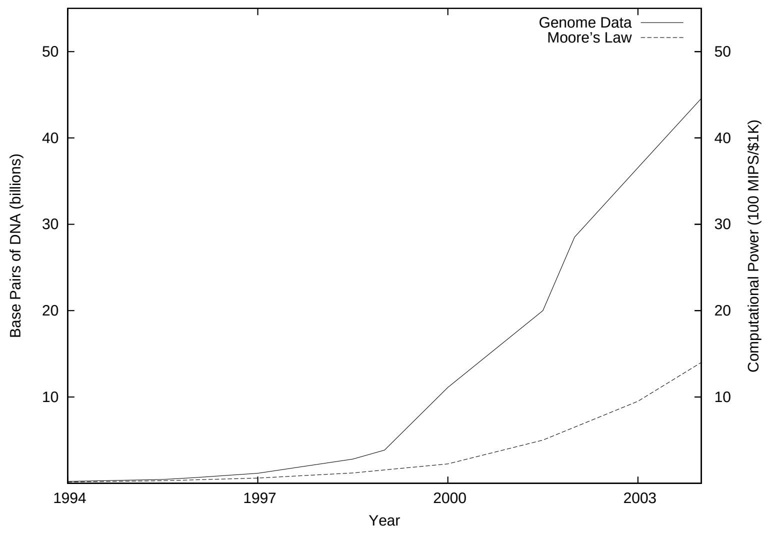


Figure 1.1: Growth of Genomics Datasets

Application	Area	Data Volume	Users
VISTA [17]	Astronomy	100 TB/year	100s
LIGO [9]	Earth Sciences	250 TB/year	100s
WCER EVP [63]	Educational Technology	500 TB/year	100s
LSST [8]	Astronomy	1000 TB/year	100s
BLAST [21]	Bioinformatics	1000 TB/year	1000s
ATLAS [16]	High Energy Physics	5000 TB/year	1000s
CMS [3]	High Energy Physics	5000 TB/year	1000s

Table 1.1: Data Requirements of Scientific Applications

scheduling [19,58,80,111], and Grid computing [6,48,59,93] to address this problem.

While existing solutions work well for compute-intensive applications, they fail to meet the needs of the data intensive applications which access, create, and move large amounts of data over wide-area networks. There has been a lot of work on remote access to data [50,88], but this approach does not work well as the target data set size increases. Moving the application closer to the data is another solution, but not always practical, since storage systems generally do not have sufficient computational resources nearby. Another approach is to move data to the application. This requires careful management of "data placement", i.e. allocating storage space and staging-in the input data before computation, and de-allocating storage space, staging-out the generated data, and cleaning up everything after the computation.

Although data placement is of great importance for the end-to-end performance of an application, current approaches closely couple data placement and computation, and consider data placement as a side effect of computation. Data placement is either embedded in the computation and causes the computation to delay, or performed as simple scripts which do not have the privileges of a job (e.g. they are not scheduled). There are even cases where the data is dumped to tapes and sent to the destination via postal services [46]. This causes inefficient and unreliable data placement due to problems such as:

1. Insufficient storage space when staging-in the input data, generating the output, and staging-out the generated data to a remote storage.
2. Trashing of storage server and subsequent timeout due to too many concurrent read data transfers.
3. Storage server crashes due to too many concurrent write data transfers.
4. Data transfers hanging indefinitely, i.e. loss of acknowledgment during third party transfers.
5. Data corruption during data transfers due to faulty hardware in the data stage and compute hosts.
6. Performance degradation due to unplanned data transfers.
7. Intermittent wide-area network outages.

In this dissertation, we propose a framework that de-couples computation and data placement, allows asynchronous execution of each, and treats data placement as a full-fledged job that can be queued, scheduled, monitored and check-pointed like computational jobs. We regard data placement as an important part of the end-to-end process, and express this in a workflow language.

As data placement jobs have different semantics and different characteristics than computational jobs, not all traditional techniques applied to computational jobs apply to data placement jobs. We analyze different scheduling strategies for data placement, and introduce a batch scheduler specialized for data placement. This scheduler implements techniques specific to queuing, scheduling, and optimization of data placement jobs, and provides a level of abstraction between the user applications and the underlying data transfer and storage resources.

We provide a complete data placement subsystem for distributed computing systems, similar to I/O subsystem in operating systems. This system offers transparent failure handling, reliable, efficient scheduling of data resources, load balancing on the storage servers, and traffic control on network links. It provides policy support, improves fault-tolerance and enables higher-level optimizations including maximizing the application throughput.

Through deployment in several real-life applications such as US-CMS, DPOSS Astronomy Pipeline, and WCER Educational Video Pipeline, our approach has proved to be effective, providing a promising new research direction.

In the remainder of this chapter, we will give examples of how data placement is handled in data intensive applications by the traditional approaches. We will then provide an historical perception on decoupling computation from I/O, and summarize the contributions of our work. We will conclude with the outline of the dissertation.

1.1 Motivating Examples

In this section, we provide two motivating examples illustrating how data placement is handled by scientific applications today. The first example is from a bioinformatics application: Blast; the second example is from a high energy physics application: CMS.

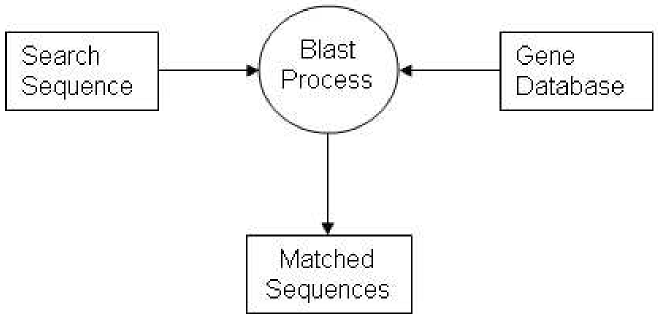


Figure 1.2: Blast Process (*blastp*)

1.1.1 Blast

Blast [21] is a bioinformatics application, which aims to decode genetic information and map genomes of different species including humankind. Blast uses comparative analysis techniques while doing this and searches for sequence similarities in protein and DNA databases by comparing unknown genetic sequences (on the order of billions) to the known ones.

Figure 1.2 shows the Blast process, the inputs it takes and the output file it generates, as observed by the traditional CPU-centric batch schedulers [101]. This is a very simplistic view of what is actually happening in a real distributed environment when we think about the end-to-end process. The diagram in Figure 1.2 does not capture how the input data is actually transferred to the execution site, and how the output data is utilized.

If we consider the end-to-end process, we see how actually the data is moved and processed by the Blast application in a distributed environment, shown in Figure 1.3. Data movement definitely complicates the end-to-end process. In Figure 1.3a, we see the script used to fetch all the files required by the Blast application, such as the executables, the gene database, and the input sequences. After all of the files are transferred to the execution site and preprocessed, a Directed Acyclic Graph (DAG) is generated, where jobs are represented as nodes and dependencies between jobs are represented as directed arcs. This DAG, shown in Figure 1.3b, can have up to n Blast processes (*blastp*) in it, all of which can be executed

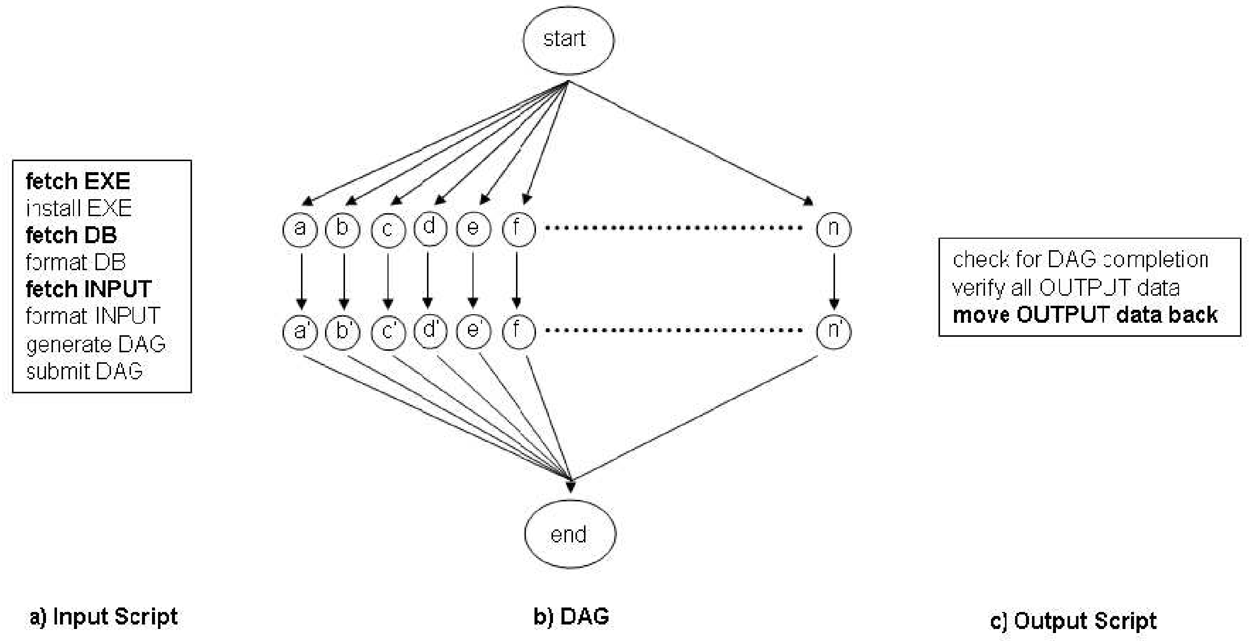


Figure 1.3: End-to-end Processing Performed by Blast

concurrently. After completion of each *blastp* process *i*, a parser process *i'* is executed which extracts the useful information from the output files generated by *blastp* and reformats them. If these two processes get executed on different nodes, the transfer of the output file from the first node to the second one is performed by the file transfer mechanism of the used batch scheduling system. When all of the processes in the DAG complete successfully, another script is executed, which is shown in Figure 1.3c. This script double-checks the generated output files for consistency and then transfers them back to the home storage.

Blast Problems

During Blast end-to-end processing, most of the data movements are handled using some scripts before and after the execution of the actual compute jobs. The remaining intermediate data movements between jobs are performed by the file transfer mechanism of the batch scheduling system used for computation. The compute jobs are scheduled by the batch scheduler for execution. On the other side, the data transfer scripts are run as "fork" jobs, generally at the head node, which do not get scheduled at all. There are no concurrency controls on the data transfers and no optimizations. Too many concurrent transfers can overload network links, trash storage servers, or even crash some of the source or destination nodes. They can fill in all of the disk space available before even a single transfer gets completed, since no space allocation is performed, which in turn can cause the failure of all of the jobs. More than one computational job on the same execution host or on the same shared file system can ask for the same files, and there can be many redundant transfers.

A message sent from one of the Blast site administrators to the rest of the Blast users illustrates this very well:

> *"... I see a lot of gridftp processes on ouhep0, and about 10 GB of new stuff in the $DATA area. That's taxing our NFS servers quite a bit (load averages of up to 40), and you're about to fill up $DATA, since that's only a 60 GB disk. Keep in mind that this is a small test cluster with many really old machines. So please don't submit any jobs which need a lot of resources, since you're likely to crash those old nodes, which has happened before ..."*

1.1.2 CMS

The Compact Muon Solenoid (CMS) [3] is a high energy physics project that aims to probe the fundamental forces in the universe by detecting proton-proton collisions at very high speeds. One of the goals is to search for the yet-undetected Higgs Boson particle [14]. There can be up to one billion proton collisions per second, generating data in the order of several petabytes per year.

The CMS processing pipeline consists of four steps: *cmkin, oscar, orca-digi,* and *orca-dst* as shown in Figure 1.4. In the first step, *cmkin* takes a random seed as an input and generates *ntuples* representing the accelerated particles. In the second step, *oscar* reads events from

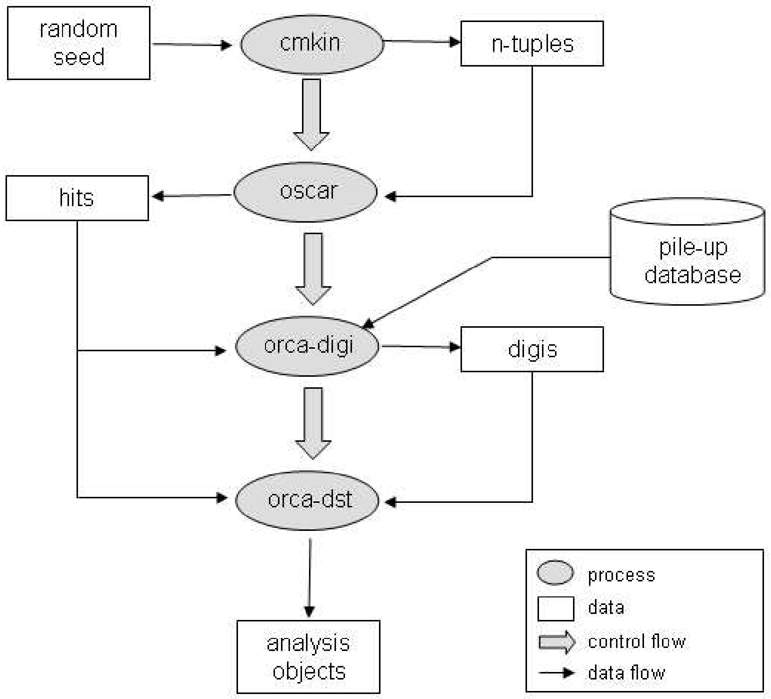


Figure 1.4: Data and Control Flow in CMS

ntuples and simulates how particles propagate through space and interact with the detector material, generating *hits* which represent this information. In the third step, *orca-digi* takes *hits* and a pile-up database as input and simulates the response of the readout electronics. These simulation results are recorded as *digis*. In the final step, *orca-dst* reads *digis* and constructs the final analysis objects.

The staging-in and staging-out of all input and output data is performed through scripts which are executed as “fork” jobs. Similar to Blast, these scripts are not scheduled by the batch scheduler, and storage space is not allocated for the data to ensure successful completion of the transfers.

CMS Problems

The jobs in the CMS pipeline ask for the transfer of the input and output data and wait until all steps (execution+transfer) finish successfully or fail. Ideally, the actual transfer should happen asynchronously to the execution. Once input or output data is asked to be transferred, the asking job should be relieved.

A message from a CMS user looking for solution for one of the problems they are facing in running CMS jobs using the classical approaches follows:

> *"... I submit 100 CMKIN jobs and throttle them 20 at a time, that will mean 100 "stage-in" completing 20 at a time. Good !? But then "runs" will go next, 20 at a time, piling up a HUGE amount of data to be "staged-out". This pile will STAY there, as long as almost all runs finish, and then will be brought back in "stage-outs". And then clean-up will happen. If during this time system goes out of disk-space, we are "dead". If system goes down, we will lose whatever we had produced during this time and several other possible disasters ..."*

1.2 A Historical Perspective

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.

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 [31].

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 [41]. 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 [84].

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. Our framework provides a clear separation of computation and data placement at the distributed systems level. A CPU-centric batch scheduler handles the scheduling and management of the computational tasks, and a data placement scheduler handles the same for the data placement tasks. This allows asynchronous execution of computational and data placement tasks. The data placement scheduler also provides a uniform interface to all underlying data transfer protocols and data storage systems. In that sense, our system resembles the I/O control system in the operating systems. We can call it a "data placement subsystem" for a distributed computing environment. With the added components for run-time adaptation, failure detection and recovery, and a feedback mechanism, it makes a complete system for reliable and efficient data placement.

1.3 Contributions

The major contributions of this dissertation are as follows:

Data Placement as a First Class Citizen. Existing systems consider data placement as a side effect of computation. In this dissertation, we introduce the concept that data placement activities must be first class citizens in a distributed computing environment just like the computational jobs [72]. They are considered as full-fledged jobs which can be queued, scheduled, monitored, and even check-pointed. We regard data placement as an important part of the end-to-end process, and express this in a workflow language.

Special Treatment to Data Placement. We introduce a framework in which data placement jobs are treated differently than computational jobs since they have different semantics and different characteristics. Not all traditional techniques applied to computational jobs apply to data placement jobs. Data placement jobs and computational jobs should be differentiated from each other and each should be submitted to specialized schedulers that understand their semantics.

A Specialized Data Placement Scheduler. We design, implement and evaluate a batch scheduler specialized in data placement: Stork [73]. This scheduler implements techniques specific to queuing, scheduling, and optimization of data placement jobs, and provides a level of abstraction between the user applications and the underlying data transfer and storage resources.

Scheduling Strategies for Data Placement. We introduce several scheduling strategies for data placement, discuss their similarities and differences compared to the scheduling strategies for computational jobs, and evaluate them.

Data Placement Subsystem. We design, implement, and evaluate a complete *data placement subsystem* [74] for distributed computing systems, similar to the I/O subsystem in operating systems. This subsystem includes the specialized scheduler for data placement (Stork), a planner aware of data placement tasks [75], a knowledgebase [62] which can extract useful information from history (log files) and interpret it, a failure detection and classification mechanism (Phoenix) [64], and some runtime optimization tools [61]. 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.

Profiling Data Transfers. We also provide a detailed profiling study of data transfer protocols and storage servers [65]. We examine the effect of some protocol parameters such as concurrency level, number of parallel streams, and I/O block size on server and client CPU load and data transfer rate. This study makes the trade-offs such as high server load versus high data transfer rate clear and makes it easier to optimize the end-to-end performance of the entire system.

1.4 Outline

The organization of the rest of this dissertation is as follows. In Chapter 2, we provide the detailed profiling study we have performed on data transfer protocols and storage servers. In Chapter 3, we introduce our data placement subsystem, and its core component: the specialized data placement scheduler (Stork). We also examine different scheduling strategies for data placement. In Chapter 4, we present how the system we have designed can dynamically adapt at runtime to changing environment. In Chapter 5, we introduce the Grid Knowledgebase, which can extract useful information from job log files, interpret it, and feed it back to the data placement subsystem. In Chapter 6, we present the failure detection and classification mechanisms we have designed. We discuss related work in Chapter 7, and conclude in Chapter 8 with a summary of the dissertation and possible future research directions.

Chapter 2

Profiling Data Transfers

2.1 Introduction

The performance and reliability of data intensive scientific applications mostly depend on how data is stored, accessed, and transferred. This fact brings the data transfer protocols and storage servers into focus. We have performed a full system profile of GridFTP [20] and NeST [12], two widely used data transfer and storage systems and provide a detailed study [65] of how CPU time is spent by each of them.

Having put the similarities and differences in CPU consumption characteristics of both systems, we focused on GridFTP, which is the most widely used data transfer protocol on the Grids today, and got a more detailed characterization of the system. We studied the effect of concurrency level and some protocol parameters such as number of parallel streams and I/O block-size on server load and transfer rate.

Some options like concurrency level increases single client performance at the expense of increased server load. This in turn affects the performance of the other clients using the same server. What ultimately matters for most scientific applications is the throughput of the entire system. Some options may increase the flexibility or security of the system at some expense. The objective of this study is to make such trade-offs clear and make it easy to optimize the entire system.

A good understanding of such trade-offs in the utilization of the data transfer protocols and storage servers helps us to make better scheduling decisions. It also helps computer architects to add processor features and operating system designers to optimize the operating systems according to the requirements of different data transfer protocols and storage systems. It enables middleware and application developers to optimize their software and helps grid deployers to choose appropriate machine configuration for their applications.

2.2 Profiling Methodology

We have selected GridFTP and NeST for our study. The reason we have selected GridFTP is that it is the most widely used data transfer protocol in the Grids today. On the other hand, we have found NeST interesting, since it supports space allocation and provides a variety of interfaces to access the data: native chirp [12], NFS [95] and GridFTP.

Our desire to perform a full-system characterization including the path through the kernel while keeping the system perturbations minimal narrowed our choice of profiler to Oprofile [13], a Linux system-wide profiler based on Digital Continuous Profiling Infrastructure [22]. Oprofile uses the hardware performance counters on the Pentium family of processors.

For profiling, we setup two server machines: a moderate server, 1660 MHz Athlon XP CPU with 512 MB RAM, and a powerful server, dual Pentium 4 2.4 GHz CPU with 1 GB RAM. Both servers used Linux kernel 2.4.20. The moderate server had 100 Mbps connectivity while the powerful one had 1 Gbps connectivity. We used three client machines, two of them were in local area and one was in wide area. The local area clients were dual 2.8 GHz Xeons and had 100 Mbps connectivity and were chosen randomly from a pool of 50 machines and the wide area client was quad 2 GHz Xeon with 100 Mbps connectivity. The powerful machines ensured that the clients were not the bottleneck and brought out the server characteristics.

We got full system profiles for both of GridFTP 2.4.3 and NeST servers using clients in the local area. For the extended study of GridFTP performance, we used clients both in local area and wide area.

Since we used real wide-area transfers we did not have any control over the loss rate. We did not trace it during the experiment because we felt such a packet trace collection at end hosts would interfere with our experiment. But we did periodic network traces and found that wide-area losses were negligible (less than 0.5%) at 100 Mbps. We had a 655 Mbps wide-area ATM connectivity and we found that the packet losses started showing up above 250 Mbps.

We tried out some commonly used options like parallel streams and concurrent number of file transfers in GridFTP and found the effect on server load.

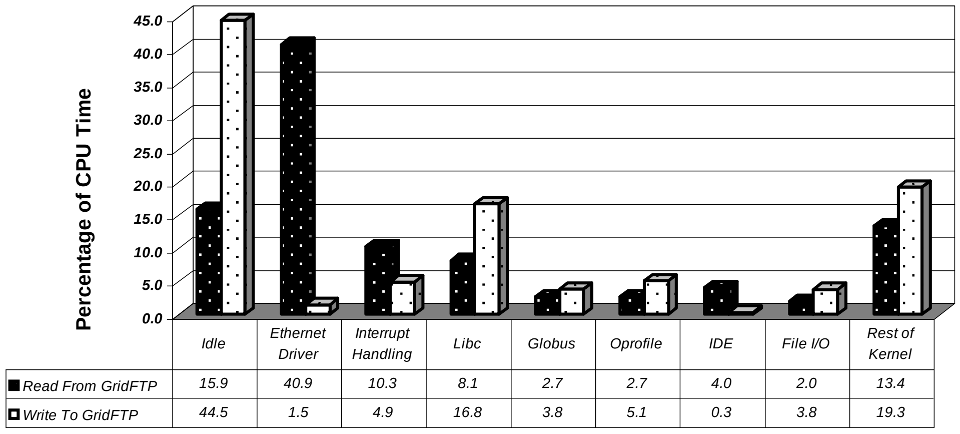
2.3 Full System Characterization

We studied how the time is spent on the server side and present the results in this section. This characterization details the fraction of time spent in the different system parts including the kernel. This is significant for data servers because most of the time is spent in the kernel and plain user-level server profile is not sufficient.

2.3.1 GridFTP

Figure 2.1 shows the GridFTP server CPU profile when a single local area client reads/writes a set of 100 MB files. The read and write clients achieved a transfer rate of 6.45 MBPS and 7.83 MBPS respectively.

In terms of server CPU load, reads from the server are more expensive than writes to the server. The extra cost is spent in interrupt handling and in the Ethernet driver. The machine has an Intel Ether Express Pro 100 network interface card(NIC). We found that



	Idle	Ethernet Driver	Interrupt Handling	Libc	Globus	Oprofile	IDE	File I/O	Rest of Kernel
■ Read From GridFTP	15.9	40.9	10.3	8.1	2.7	2.7	4.0	2.0	13.4
□ Write To GridFTP	44.5	1.5	4.9	16.8	3.8	5.1	0.3	3.8	19.3

Figure 2.1: GridFTP Server CPU Profile

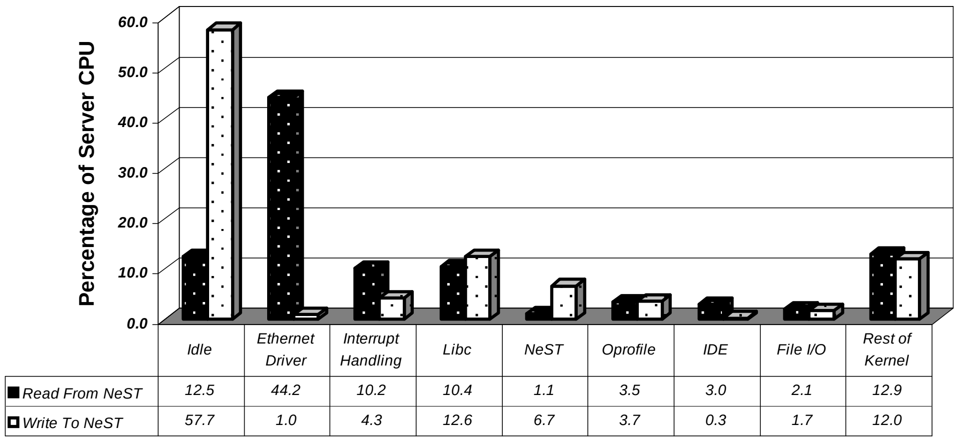
interrupt coalescing lowered the interrupt cost during write to server. The NIC transfers the received packets via DMA to main memory resulting in low CPU cost for writes to server. CPU is used to transfer output packets to the NIC resulting in high cost of read from the server. NIC with capability to DMA the output packets along with a driver capable of using that feature would reduce server read load considerably.

In the Libc, 65% of the time is spent in the getc function. The IDE disk has a greater overhead on reads compared to writes. Tuning the disk elevator algorithm may help here. The file I/O part includes the time spent in the filesystem. It is higher for writes because of the need for block allocation during writes. The rest of the kernel time is spent mostly for TCP/IP, packet scheduling, memory-copy, kmalloc and kfree.

2.3.2 NeST

Figure 2.2 shows the NeST server CPU profile when a single local area client reads/writes a set of 100 MB files using the chirp protocol. The read and write clients achieved a transfer rate of 7.49 MBPS and 5.5 MBPS respectively.

NeST server has a 16% higher read transfer rate and a 30% lower write transfer rate compared to GridFTP server. The lower performance of writes to NeST server is because



	Idle	Ethernet Driver	Interrupt Handling	Libc	NeST	Oprofile	IDE	File I/O	Rest of Kernel
■ Read From NeST	12.5	44.2	10.2	10.4	1.1	3.5	3.0	2.1	12.9
□ Write To NeST	57.7	1.0	4.3	12.6	6.7	3.7	0.3	1.7	12.0

Figure 2.2: NeST Server CPU Profile

of the space allocation feature called 'Lots'. Before each write, NeST server verifies that the client has not exceeded the storage allocation, and at the end of write, it updates this meta-data persistently. This causes the slowdown. NeST allows turning off 'Lots' and doing that makes the write performance close to that of GridFTP server. This shows that space allocation, while being a useful feature, comes with a certain overhead.

2.4 Effect of Protocol Parameters

GridFTP allows us to use different block-sizes and multiple parallel streams. Further, clients can concurrently transfer multiple files to/from the server. We studied the effect of the above parameters and concurrency on transfer rate and CPU utilization.

The effect of using different block-sizes and parallelism while writing to the moderate GridFTP server is shown in Fig. 2.3. Interestingly, the server load drops and the transfer rate increases as we move from one stream to two streams. We analyzed further and decided to look at the Translation Look-Aside Buffer(TLB) misses. TLB is a cache that speeds up translating virtual addresses to physical addresses in the processor. As seen in Fig. 2.3c, the L2 Data TLB misses drops as the parallelism is increased from one to two. The drop in L2

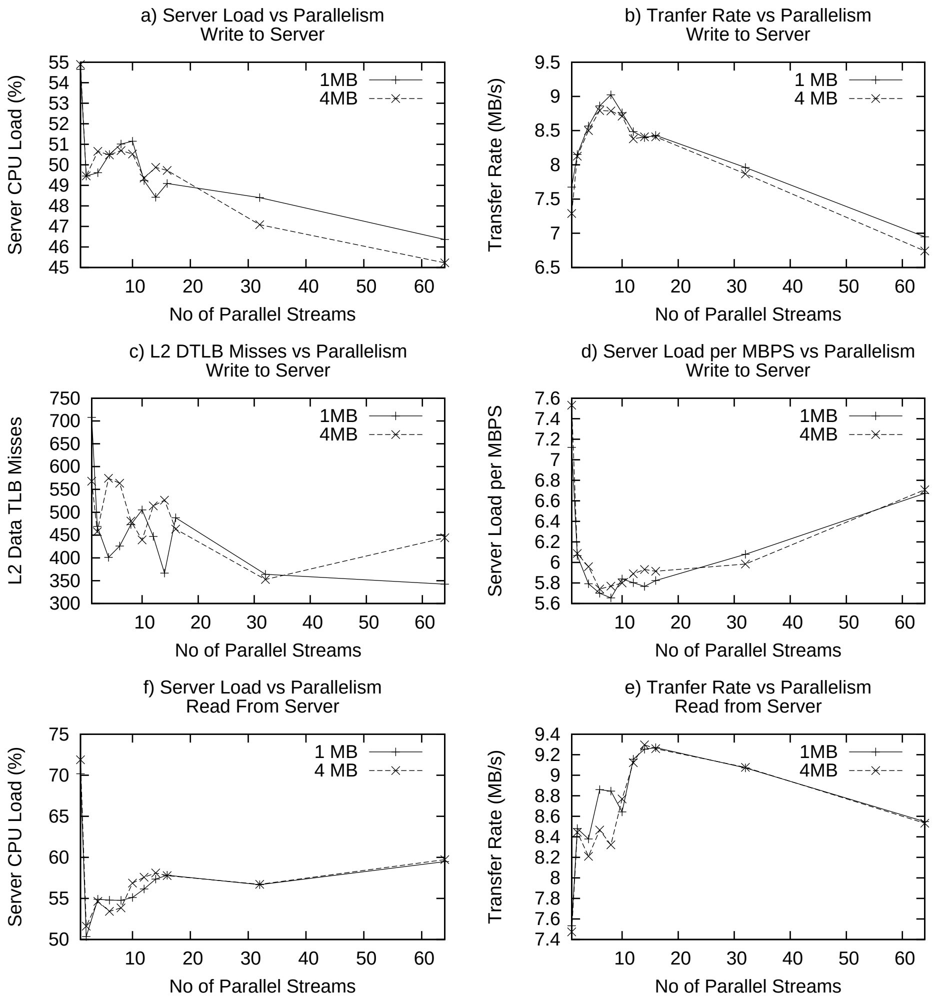


Figure 2.3: Effect of Block Size and Parallelism

DTLB misses explains the simultaneous decrease in server load and increase in transfer rate.

We went a step further and tried to find out what was causing the reduction in L2 DTLB misses and found that the Pentium processor family supports a large page size of 4 MB in addition to the normal page size of 4 KB. For data servers, using the large pages would be greatly beneficial. Unfortunately, the Linux kernel at present does not allow application to request such jumbo pages, but internally the kernel can use these large pages. We found that the internal kernel usage of jumbo 4 MB pages during use of parallel streams causes the drop in TLB misses. We also found that using a block size of 4 MB did not make the kernel use the jumbo page internally.

We tried the experiment with different machines and found that they had a different parallelism TLB miss graph. The variance in TLB misses was quite small until 10 parallel streams and starts rising afterwards. Another interesting result we found was that the TLB miss graph of a machine at different times was similar. At present, it appears that the Linux kernel usage of large pages internally depends mostly on the machine configuration. This requires a more thorough analysis.

Figure 2.3d shows the server load per MBPS transfer rate. Data movers may want to lower server CPU load per unit transfer rate and this graphs shows how they can use parallelism to achieve this.

The effect of block size when reading from the server is shown in Fig. 2.3e and 2.3f. We find that the optimal parallelism for reading from the server is different from that used to write to it.

We have studied the effects of different concurrency and parallelism levels on the transfer rate and CPU utilization. This study was done using the powerful server and the effect on write to server is shown in Fig. 2.4. We observed that increasing the number of concurrent files being transferred results in a higher increase in the transfer rate compared to increasing the number of parallel streams used to transfer a single file (Fig. 2.4a and 2.4b). This is the result of multiple transfers being able to saturate the bandwidth better than single transfer with multiple streams. In wide area, both increasing the level of concurrency and parallelism improve the performance considerably (Fig. 2.4b). Whereas in the local area both have very little positive impact on the performance, and even cause a decrease in the transfer rate for slightly high concurrency and parallelism levels (Fig. 2.4a). As the file size increases, the impact of parallelism level on transfer rate increases as well (Fig. 2.4c). This study shows that increased parallelism or concurrency level does not necessarily result in better performance, but depends on many parameters. The users should select the correct

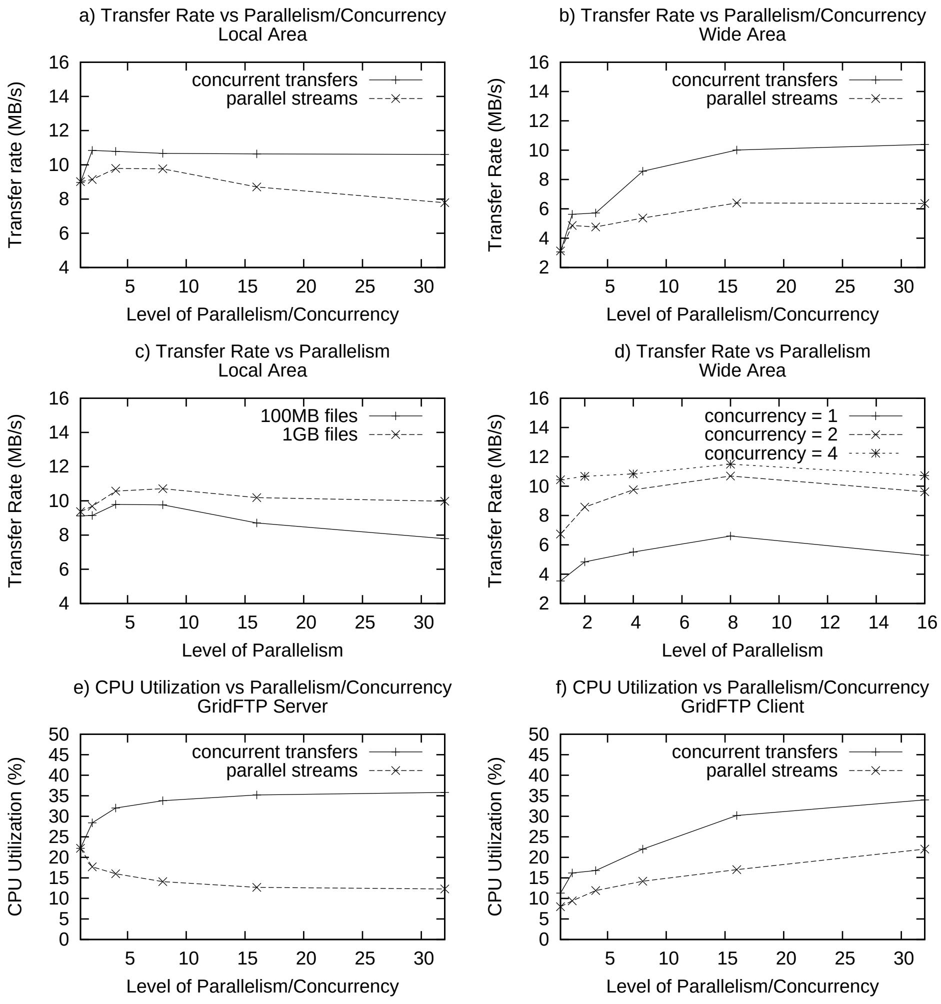


Figure 2.4: Effect of Concurrency and Parallelism

parallelism or concurrency level specific to their settings for better performance.

Increasing the concurrency level also results in a higher load on the server (Fig. 2.4e), whereas increasing the number of parallel streams decreases server CPU utilization. As stated by the developers of GridFTP, a reason for this can be the amortization of the select() system call overhead. The more parallel streams that are monitored in a select() call, the more likely it will return with ready file descriptors before GridFTP needs to do another select() call. Also, as the more file descriptors that are ready, the more time is spent actually reading or writing data before the next select() call. On the client side, both increasing the concurrency and parallelism levels cause an increase in the CPU utilization of the client machine (Fig. 2.4f).

We believe that using a combination of concurrency and parallelism can result in higher performance than using parallelism only or concurrency only (Fig. 2.4d). It can also help in achieving the optimum transfer rate by causing lower load to the server.

2.5 Discussion

In this work, we have performed a full system profile of GridFTP and NeST, two widely used data access and storage systems, and provided a detailed study of how time is spent in each of their servers.

We have characterized the effect of concurrency level and GridFTP protocol parameters such as I/O block size and number of parallel streams on data transfer rate and server CPU utilization. We have made clear the trade-off between single client performance and server load and shown how client performance can be increased and server load decreased at the same time and explained the reason behind this. This allows users to configure and optimize their systems for better end-to-end transfer performance and higher throughput.

This study can be extended further in order to find better correlations between different parameters. This will help us to provide more useful information to users helping them to perform more sophisticated configuration and optimization.

The results of this study can be used by the batch schedulers, as presented in the following chapters, in order to make better scheduling decisions specific to the needs of particular data transfer protocols or data storage servers. This will allow us to increase the throughput of the whole system, and decrease the response time for the individual applications. Controlling the CPU server load can also increase reliability of the whole system by preventing server crashes due to overloads.

Chapter 3

Scheduling Data Placement

3.1 Introduction

In Chapter 1, we mentioned the increasing and overwhelming data requirements of scientific applications. 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, the number of people who will be accessing the datasets range from 100s to 1000s, as shown in Table 1.

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 the privileges of a job, they do not get scheduled, 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 [46].

Data placement activities must be first class citizens in the distributed computing environments just like the computational 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.

Moreover, data placement jobs should be treated differently from computational jobs, since they have different semantics and different characteristics. 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, data placement jobs and computational jobs should be differentiated from each other and each should be submitted to specialized schedulers that understand their semantics.

We have designed and implemented the first batch scheduler specialized for data placement: Stork [73]. This scheduler implements techniques specific to queuing, scheduling, and optimization of data placement jobs, and provides a level of abstraction between the user applications and the underlying data transfer and storage resources.

We also provide a complete "data placement subsystem" [75] [74] for distributed computing systems, similar to I/O subsystem in operating systems. This subsystem includes the specialized scheduler for data placement (Stork), a higher level planner aware of data placement jobs [75], a knowledgebase [62] which can extract useful information from history (log files) and interpret it, a failure detection and classification mechanism (Phoenix) [64], and some runtime optimization tools [61]. 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.

3.2 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 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 3.1.

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 [40], Chimera [51]) 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 [105]). The workflow manager submits computational jobs to a compute job queue, and the data placement jobs to a data

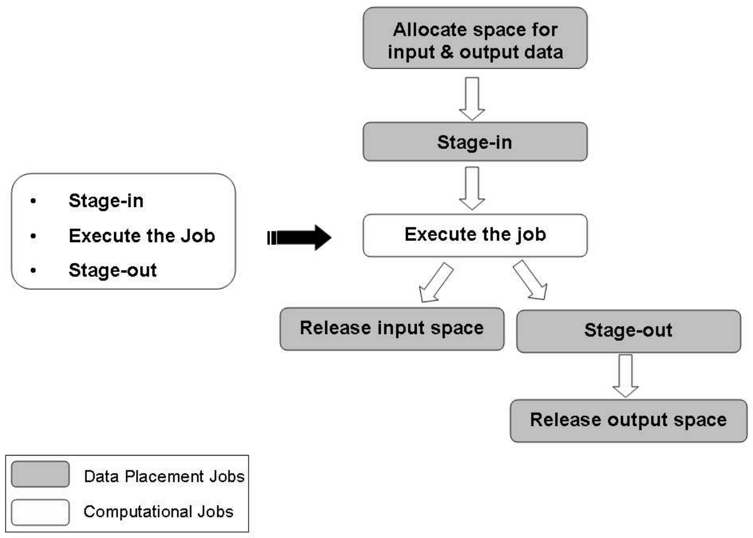


Figure 3.1: Separating Data Placement from Computation

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 approaches, 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 [37]) 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 network monitoring tools collect statistics on maximum available end-to-end bandwidth, actual bandwidth utilization, latency and number of hops to be traveled by utilizing tools such as Pathrate [44] and Iperf [15]. Again, the collected statistics are fed back to the scheduler and the resource broker/policy enforcer. The design and implementation of our network monitoring tools, and how they are used to dynamically adapt data placement jobs to the changing environment are presented in Chapter 4.

The log files of the jobs are collected by the knowledgebase. The knowledgebase 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 in the knowledgebase. The knowledgebase 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 design and implementation of our prototype knowledgebase is presented in Chapter 5

The failure agent is used to detect and classify failure as early as possible from the information gathered from the log files and also from the feedback retrieved from the knowledgebase. The design and implementation of our prototype failure agent is presented in Chapter 6.

The components of our data placement subsystem and their interaction with other components are shown in Figure 3.2. 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.

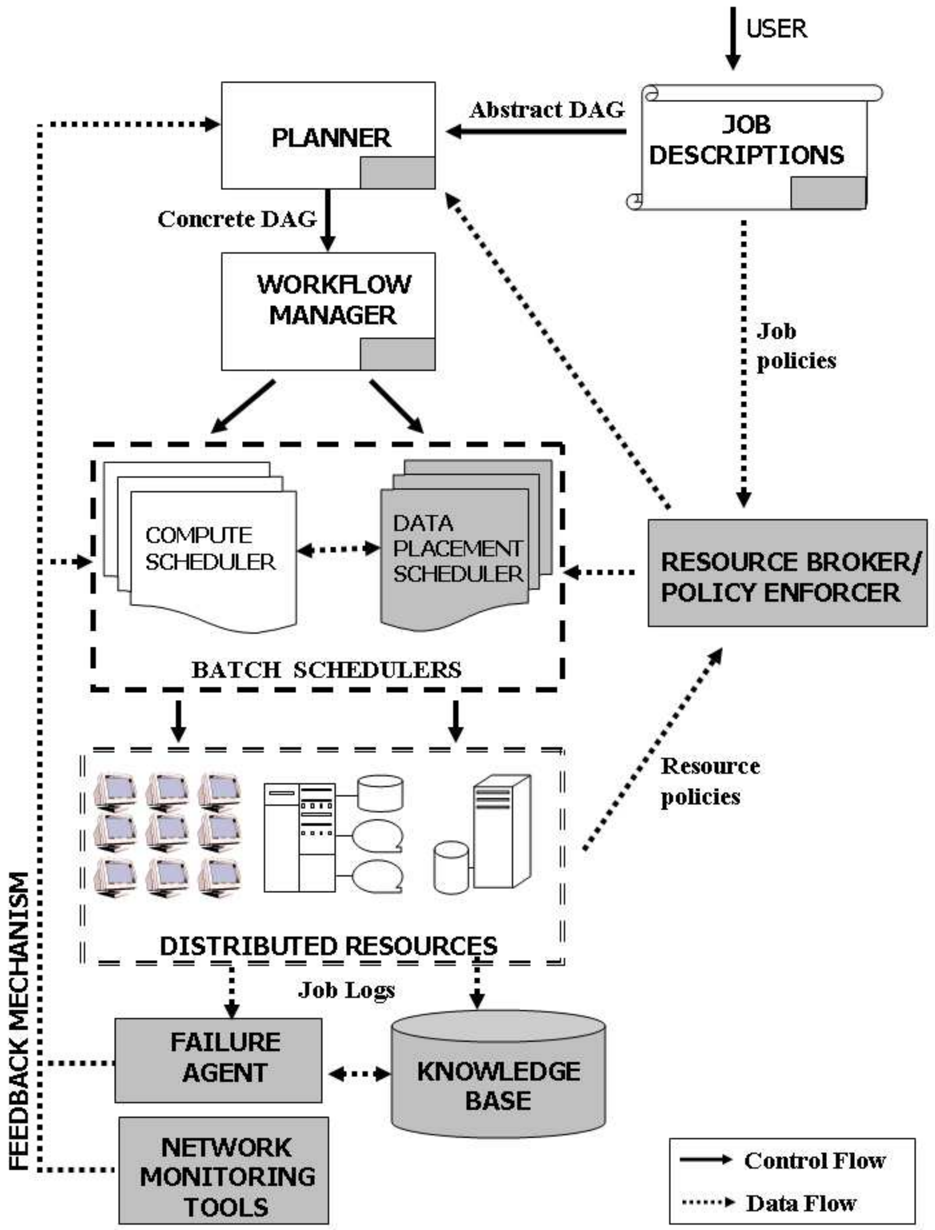


Figure 3.2: Components of the Data Placement Subsystem

3.3 Data Placement Scheduler (Stork)

We have designed, implemented and evaluated the first batch scheduler specialized in data placement: Stork [73]. This scheduler implements techniques specific to queuing, scheduling, and optimization of data placement jobs, and provides a level of abstraction between the user applications and the underlying data transfer and storage resources.

3.3.1 Data Placement Job Types

We categorize data placement jobs into seven types. These are:

transfer: This job type is for transferring a complete or partial file from one physical location to another one. This can include a get or put operation or a third party transfer.

allocate: This job type is used for allocating storage space at the destination site, allocating network bandwidth, or establishing a light-path on the route from source to destination. Basically, it deals with all necessary resource allocations pre-required for the placement of the data.

release: This job type is used for releasing the corresponding resource which is allocated before.

remove: This job is used for physically removing a file from a remote or local storage server, tape or disk.

locate: Given a logical file name, this job consults a meta data catalog service such as MCAT [27] or RLS [37] and returns the physical location of the file.

register: This type is used to register the file name to a meta data catalog service.

unregister: This job unregisters a file from a meta data catalog service.

The reason that we categorize the data placement jobs into different types is that all of these types can have different priorities and different optimization mechanisms.

3.3.2 Flexible Job Representation and Multilevel Policy Support

Stork uses the ClassAd [92] 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.

Below, you see three sample data placement (DaP) requests:

```
[
  dap_type   = ``allocate'';
  dest_host  = ``turkey.cs.wisc.edu'';
  size       = ``200MB''';
  duration   = ``60 minutes'';
  allocation_id = 1;
]

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

[
  dap_type  = ``release'';
  dest_host = ``turkey.cs.wisc.edu'';
  allocation_id = 1;
]
```

The first request is to allocate 200 MB of disk space for 1 hour on a NeST server. The second request is to transfer a file from an SRB server to the allocated space on the NeST server. The third request is to de-allocate the previously allocated space.

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'';
]
```

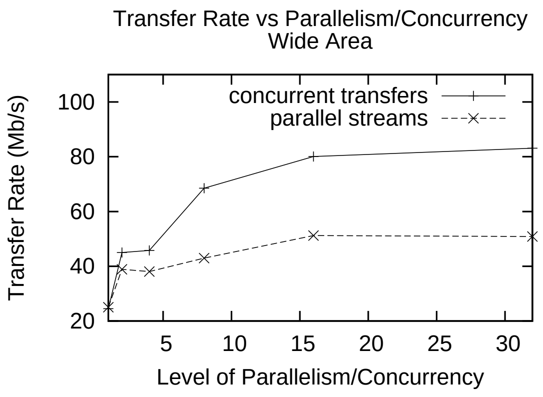
In this example, the user specifies that the job should be retried up to 10 times in case of failure, and if the transfer does not get completed in 2 hours, it should be killed and restarted.

3.3.3 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 allocations are supported by Stork as long as the corresponding storage systems have support for it.

At this point, it will be very useful to revisit some of the results from our profiling study in Chapter 2, and discuss how this information can be used by a data placement scheduler to make smart scheduling decisions. Figure 3.3 shows the effect of increased parallelism and concurrency levels on the transfer rate. With the level of parallelism, we refer to the number of parallel streams used during the transfer of a single file; and with the level of concurrency, we refer to the number of files being transferred concurrently.

When the level parallelism and concurrency increases, the transfer rate incurred in the wide area transfers increases as expected. But for the local area transfers, the case is different. We observe that increased parallelism and concurrency levels help with increasing the transfer



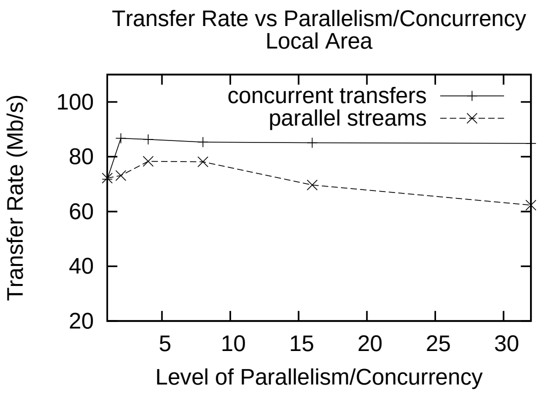


Figure 3.3: Controlling the Throughput

rate in local area transfers up to a certain point, but after that, they have a negative impact on the transfer rate. The transfer rate comes to a threshold, and after this point the overhead of using multiple streams and issuing multiple transfers starts causing a decrease in the transfer rate.

These observations show us that increasing parallelism and concurrency levels do not always increase the transfer rate. The effect on the wide and local area can be different. Increased concurrency has a more positive impact on the transfer rate compared with increased parallelism.

Figure 3.4 shows the effect of increased parallelism and concurrency levels on the CPU utilization. While the number of parallel streams and the concurrency level increases, the CPU utilization at the client side increases as expected. On the server side, same thing happens as the level of concurrency increases. But, we observe a completely opposite effect on the server side as the level of parallelism increases. With the increased parallelism level, the server CPU utilization starts dropping and keeps this behavior as long as the parallelism level is increased.

The most interesting observation here is that concurrency and parallelism have completely opposite impacts on CPU utilization at the server side. The reasoning behind this is

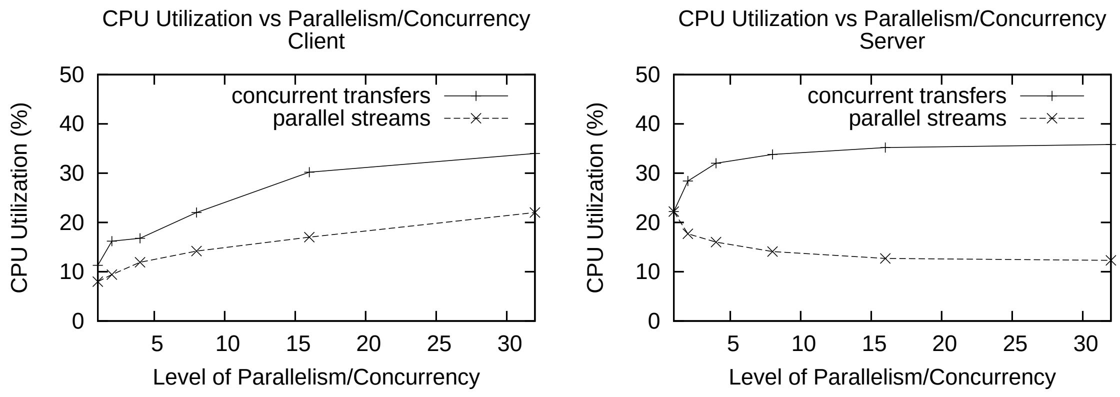


Figure 3.4: Controlling the CPU Utilization

explained in Chapter 2.

These results show that some of the assumptions we take for granted may not always hold. We need a more complicated mechanism to decide the correct concurrency or parallelism level in order to achieve high transfer rate and low CPU utilization at the same time.

3.3.4 Job Scheduling Techniques

We have applied some of the traditional job scheduling techniques common in computational job scheduling to the scheduling of data placement jobs:

First Come First Served (FCFS) Scheduling: Regardless of the type of the data placement job and other criteria, the job that has entered into the queue of the data placement scheduler first is executed first. This technique, being the simplest one, does not perform any optimizations at all.

Shortest Job First (SJF) Scheduling: The data placement job which is expected to take least amount of time to complete will be executed first. All data placement jobs except the transfer jobs have job completion time in the order of seconds, or minutes in the worst case. On the other hand, the execution time for the transfer jobs can vary from couple of seconds to couple of hours even days. Accepting this policy would mean non-transfer jobs would be executed always before transfer jobs. This may cause big delays in executing the actual transfer jobs, which defeats the whole purpose of scheduling data placement.

Multilevel Queue Priority Scheduling: In this case, each type of data placement job is sent to separate queues. A priority is assigned to each job queue, and the jobs in the highest priority queue are executed first. To prevent starvation of the low priority jobs, the traditional aging technique is applied. The hardest problem here is determining the priorities of each data placement job type.

Random Scheduling: A data placement job in the queue is randomly picked and executed without considering any criteria.

Auxiliary Scheduling of Data Transfer Jobs

The above techniques are applied to all data placement jobs regardless of the type. After this ordering, some job types require additional scheduling for further optimization. One such type is the data transfer jobs. The transfer jobs are the most resource consuming ones. They consume much more storage space, network bandwidth, and CPU cycles than any other data placement job. If not planned well, they can fill up all storage space, trash and even crash servers, or congest all of the network links between the source and the destination.

Storage Space Management. One of the important resources that need to be taken into consideration when making scheduling decisions is the available storage space at the destination. The ideal case would be the destination storage system support space allocations, as in the case of NeST [12], and before submitting a data transfer job, a space allocation job is submitted in the workflow. This way, it is assured that the destination storage system will have sufficient available space for this transfer.

Unfortunately, not all storage systems support space allocations. For such systems, the data placement scheduler needs to make the best effort in order not to over-commit the storage space. This is performed by keeping track of the size of the data transferred to, and

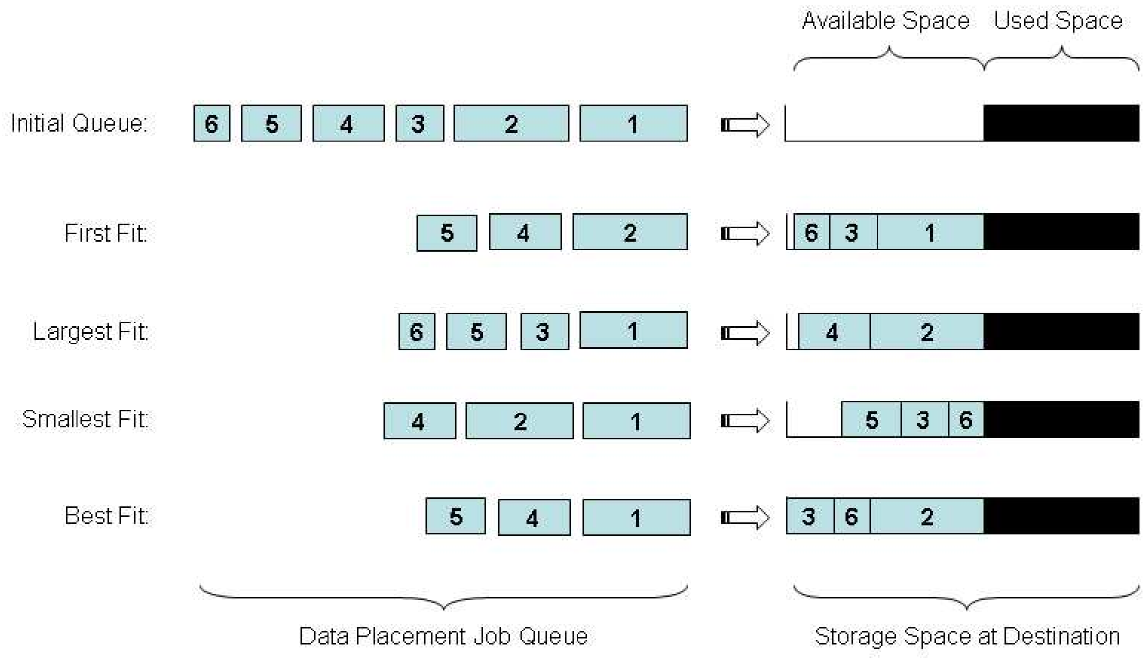


Figure 3.5: Storage Space Management: Different Techniques

removed from each storage system which does not support space allocation. When ordering the transfer requests to that particular storage system, the remaining amount of available space, to the best of the scheduler's knowledge, is taken into consideration. This method does not assure availability of storage space during the transfer of a file, since there can be external effects, such as users which access the same storage system via other interfaces without using the data placement scheduler. In this case, the data placement scheduler at least assures that it does not over-commit the available storage space, and it will manage the space efficiently if there are no external effects.

Figure 3.5 shows how the scheduler changes the order of the previously scheduled jobs to meet the space requirements at the destination storage system. In this example, four different techniques are used to determine in which order to execute the data transfer request without over-committing the available storage space at the destination: *first fit*, *largest fit*, *smallest fit*, and *best fit*.

First Fit: In this technique, if the next transfer job in the queue is for data which will not fit in the available space, it is skipped for that scheduling cycle and the next available transfer job with data size less than or equal to the available space is executed instead. It is important to point that a complete reordering is not performed according to the space requirements. The initial scheduling order is preserved, but only requests which will not satisfy the storage space requirements are skipped, since they would fail anyway and also would prevent other jobs in the queue from being executed.

Largest Fit and *Smallest Fit:* These techniques reorder all of the transfer requests in the queue and then select and execute the transfer request for the file with the largest, or the smallest, file size. Both techniques have a higher complexity compared with the *first fit* technique, although they do not guarantee better utilization of the remote storage space.

Best Fit: This technique involves a greedy algorithm which searches all possible combinations of the data transfer requests in the queue and finds the combination which utilizes the remote storage space best. Of course, it comes with a cost, which is a very high complexity and long search time. Especially in the cases where there are thousands of requests in the queue, this technique would perform very poorly.

Using a simple experiment setting, we will display how the built-in storage management capability of the data placement scheduler can help improving both overall performance and reliability of the system. The setting of this experiment is shown in Figure 3.6.

In this experiment, we want to process 40 gigabytes of data, which consists of 60 files each between 500 megabytes and 1 gigabytes. First, the files need to be transferred from the remote storage site to the staging site near the compute pool where the processing will be done. Each file will be used as an input to a separate process, which means there will be 60 computational jobs followed by the 60 transfers. The staging site has only 10 gigabytes of storage capacity, which puts a limitation on the number of files that can be transfered and processed at any time.

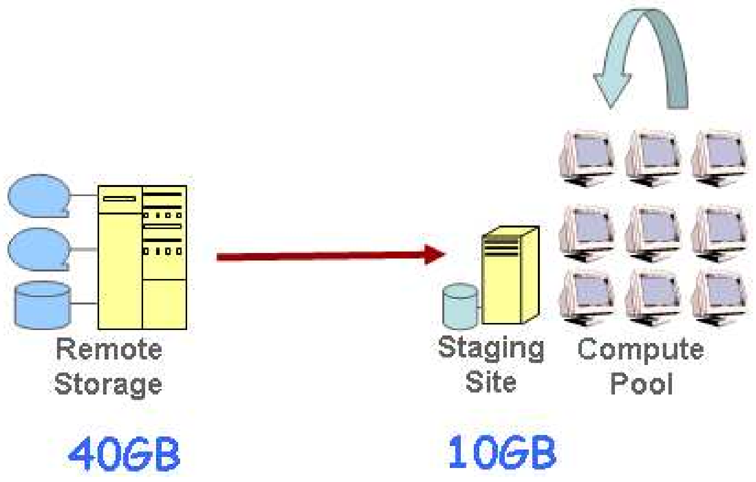


Figure 3.6: Storage Space Management: Experiment Setup

A traditional scheduler would simply start all of the 60 transfers concurrently since it is not aware of the storage space limitations at the destination. After a while, each file would have around 150 megabytes transferred to the destination. But suddenly, the storage space at the destination would get filled, and all of the file transfers would fail. This would follow with the failure of all of the computational jobs dependent on these files.

On the other hand, Stork completes all transfers successfully by smartly managing the storage space at the staging site. At any time, no more than the available storage space is committed, and as soon as the processing of a file is completed, it is removed from the staging area to allow transfer of new files. The number of transfer jobs running concurrently at any time and the amount of storage space committed at the staging area during the experiment are shown in Figure 3.7 on the left side.

In a traditional batch scheduler system, the user could intervene, and manually set some virtual limits to the level of concurrency the scheduler can achieve during these transfers. For example, a safe concurrency limit would be the total amount of storage space available at the staging area divided by the size of the largest file that is in the request queue. This would assure the scheduler does not over-commit remote storage. Any concurrency level higher

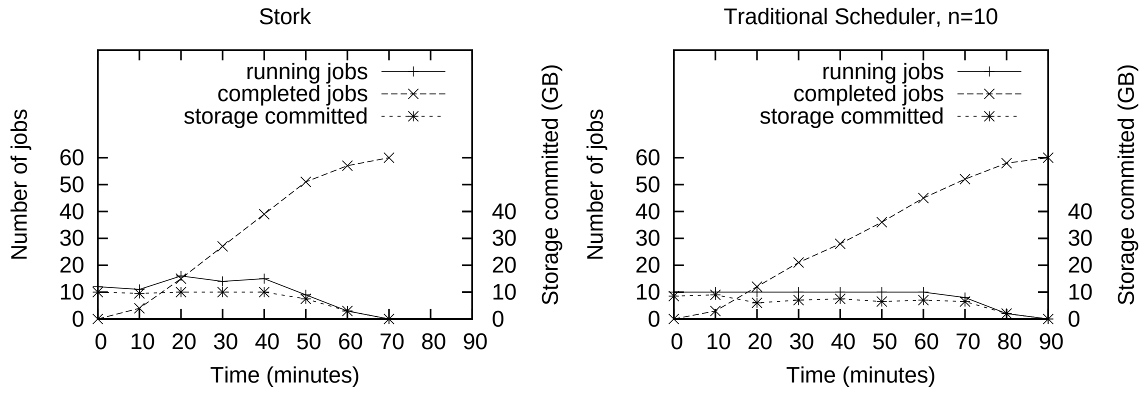


Figure 3.7: Storage Space Management: Stork vs Traditional Scheduler

than this would have the risk of getting out of disk space anytime, and may cause failure of at least some of the jobs. The performance of the traditional scheduler with concurrency level set to 10 manually by the user in the same experiment is shown in Figure 3.7 on the right side.

Manually setting the concurrency level in a traditional batch scheduling system has three main disadvantages. First, it is not automatic, it requires user intervention and depends on the decision made by the user. Second, the set concurrency is constant and does not fully utilize the available storage unless the sizes of all the files in the request queue are equal. Finally, if the available storage increases or decreases during the transfers, the traditional scheduler cannot re-adjust the concurrency level in order to prevent overcommitment of the decreased storage space or fully utilize the increased storage space.

Storage Server Connection Management. Another important resource which needs to be managed carefully is the number of concurrent connections made to specific storage servers. Storage servers being thrashed or getting crashed due to too many concurrent file transfer connections has been a common problem in data intensive distributed computing.

In our framework, the data storage resources are considered first class citizens just like the computational resources. Similar to computational resources advertising themselves, their attributes and their access policies, the data storage resources advertise themselves, their attributes, and their access policies as well. The advertisement sent by the storage resource includes the number of maximum concurrent connections it wants to take anytime. It can also include a detailed breakdown of how many connections will be accepted from which client, such as "maximum n GridFTP connections, and "maximum m HTTP connections".

This throttling is in addition to the global throttling performed by the scheduler. The scheduler will not execute more than lets say x amount of data placement requests at any time, but it will also not send more than y requests to server a, and more than z requests to server b, $y+z$ being less than or equal to x.

Other Scheduling Optimizations. In some cases, two different jobs request the transfer of the same file to the same destination. Obviously, all of these request except one are redundant and wasting computational and network resources. The data placement scheduler catches such requests in its queue, performs only one of them, but returns success (or failure depending on the return code) to all of such requests. We want to highlight that the redundant jobs are not canceled or simply removed from the queue. They still get honored and the return value of the actual transfer is returned to them. But, no redundant transfers are performed.

In some other cases, different requests are made to transfer different parts of the same file to the same destination. These type of requests are merged into one request, and only one transfer command is issued. But again, all of the requests get honored and the appropriate return value is returned to all of them.

3.3.5 Interaction with Higher Level Planners

Stork can interact with higher level planners and workflow managers. This allows the users to schedule both CPU resources and storage resources together. We have introduced a

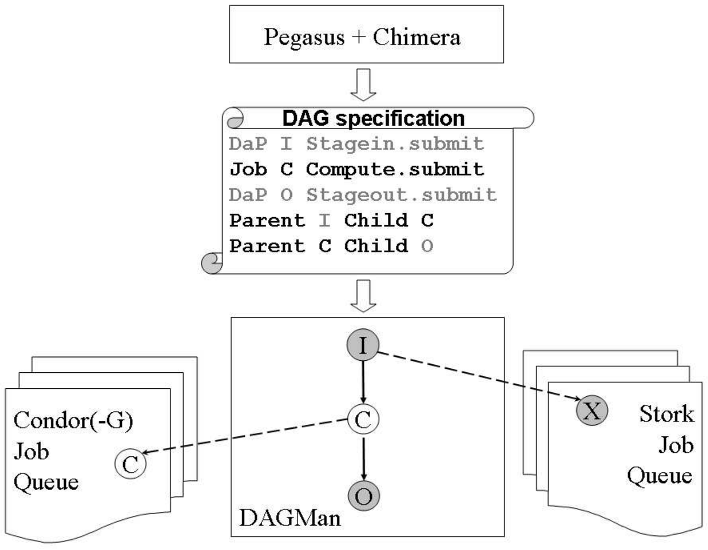


Figure 3.8: Interaction with Higher Level Planners

new workflow language capturing the data placement jobs in the workflow as well. We have also 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 [80] or Condor-G [52], and the data placement jobs to Stork. Figure 3.8 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 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.

The space allocations, stage-in and stage-outs happen asynchronously to the execution. The computational jobs do not ask for the transfer of the input and output data and wait until all steps (execution+transfer) are finished successfully or failed, which is the case in most of the traditional systems.

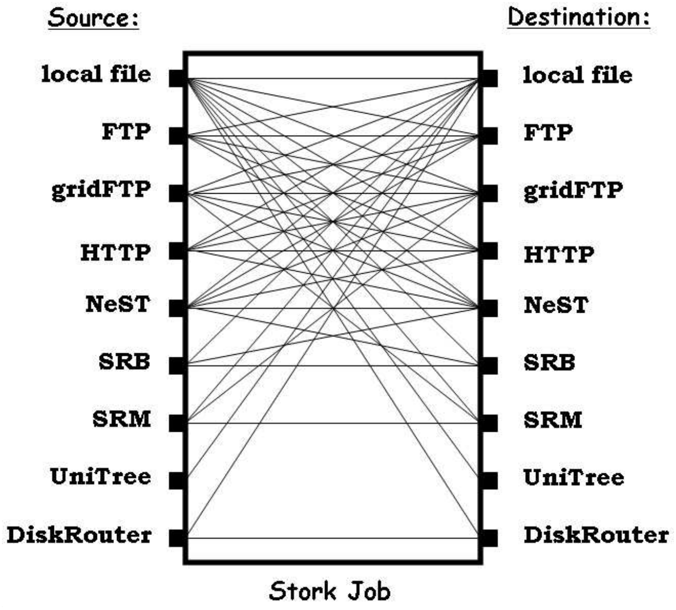


Figure 3.9: Protocol Translation using Stork Memory Buffer

3.3.6 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

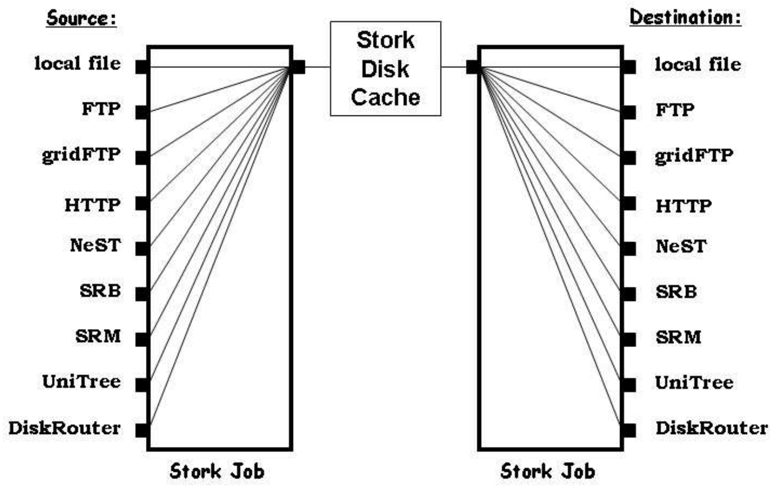


Figure 3.10: Protocol Translation using Stork Disk Cache

interact currently with data transfer protocols such as FTP [91], GridFTP [20], HTTP and DiskRouter [67]; data storage systems such as SRB [27], UniTree [33], and NeST [12]; and data management middleware such as SRM [97].

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 3.9 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 storage system. This is shown in Figure 3.10.

3.3.7 Runtime Adaptation

Sophisticated protocols developed for wide area data transfers like GridFTP allow tuning depending on the environment to achieve the best performance. While tuning by itself is difficult, it is further complicated by the changing environment. The parameters which are optimal at the time of job submission, may no longer be optimal at the time of execution. The best time to tune the parameters is just before execution of the data placement job. Determining the environment characteristics and performing tuning for each job may impose a significant overhead. Ideally, we need an infrastructure that detects environmental changes and performs appropriate tuning and uses the tuned parameters for subsequent data placement jobs.

We have designed and developed a monitoring and tuning infrastructure for this purpose. The monitoring infrastructure determines the environment characteristics and detects any subsequent change. The environment characteristics are used by the tuning infrastructure to generate tuned parameters for the various protocols. These tuned parameters are fed to Stork. Stork uses the tuned parameters while executing the data placement jobs submitted to it, essentially performing run-time adaptation of data placement jobs. Stork also has a dynamic protocol selection and alternate protocol fall-back capability. Dynamic protocol selection determines the protocols that are available on a particular host and uses an appropriate protocol for data transfer between any two hosts. Alternate protocol fall-back allows the data placement scheduler to switch to a different protocol if the protocol being used for a transfer stops working. The runtime adaptation feature will be discussed in detail in Chapter 4.

3.3.8 Learning from History

During our experience with real life data intensive distributed applications, we found the presence of "black holes", hosts that accept jobs but do not do anything for a long time. The scheduler had to try unsuccessfully to talk to the host for over several hours, and in some cases for several days. We also found cases where certain hosts had problems with certain job classes while they executed others successfully.

Detecting the above kinds of problems is difficult and the only party affected is the job that was unable to complete successfully. Further, especially in a Grid environment, the job submitter may not have control over the resource configuration. Following the 'dependability from client side' argument [82] [103], the job should be adapted to avoid those resources.

To enable this, we propose the concept of a Grid Knowledgebase that aggregates the experience of the different jobs. It collects this information from the job log files produced by the batch scheduling systems like Condor/Condor-G and Stork. These log files are normally available to the client and are different from cluster/pool log files that many site administrators are unwilling to share. This log files essentially contain the view of the world as seen by the client.

We extract useful information from the log files and enter it into a database. We add an adaptation layer that uses this collected wisdom to adapt the failed job instances and future job instances of a Grid application. This is similar to organisms learning from experience and works well because many Grid applications consist of multiple instances of the same executable operating on different data.

This information is then passed to Stork and other batch schedulers, which enables the schedulers to make better scheduling decisions in the future by using the history information. The Grid Knowledgebase will be discussed in detail in Chapter 5.

3.3.9 Failure Detection and Recovery

A major hurdle facing data intensive applications is appropriate handling of failures that occur in a Grid environment. Most application developers are unaware of the different types of failures that may occur in such an environment. Understanding and handling failures imposes an undue burden on the application developer already burdened with the development of their complex distributed application.

We feel that the middleware should tolerate faults and make this functionality transparent to the application. This would enable different data-intensive applications to become fault-tolerant without each having to pay a separate cost. Removing the burden of understanding and handling failures lets application developers concentrate on the problem at hand and reduces the time to solve their problem.

A transparent middleware solution we have designed and implemented, Phoenix [64] together with Stork, adds fault-tolerance to data intensive applications by detecting failures early, classifying failures into transient and permanent, and handling each transient failure appropriately. This solution will be discussed in detail in Chapter 6.

3.4 Case Studies

We will now show the applicability and contributions of our data placement subsystem with two case studies. The first case study shows using Stork to create a data-pipeline [69] [71] between two heterogeneous storage systems in order to transfer the three terabyte DPOSS [43] Astronomy dataset. 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 [70]. The second case study shows how our data placement subsystem can be used for to process terabytes of educational video in the WCER pipeline [63].

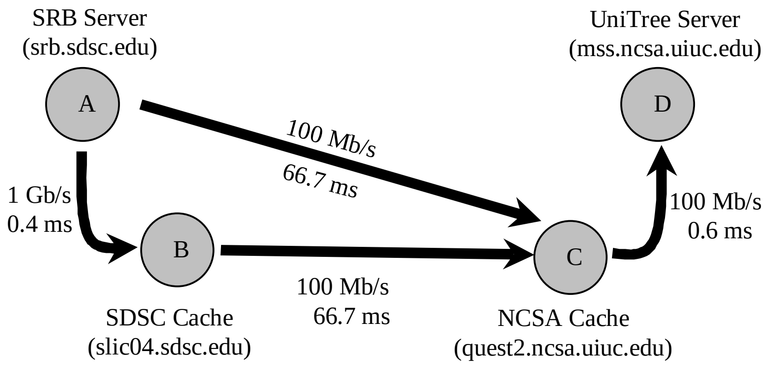


Figure 3.11: The Topology of the Network

3.4.1 DPOSS Astronomy Pipeline

National Center for Supercomputing Applications (NCSA) scientists wanted to replicate the Digital Palomar Sky Survey (DPOSS) [43] image data residing on SRB mass storage system at San Diego Supercomputing Center (SDSC) in California to UniTree mass-storage system at NCSA, Illinois to enable them to perform later processing. The total data size was around three terabytes (2611 files of 1.1 GB each). Since there was no direct interface between SRB and UniTree at the time of the experiment, the only way to perform the data transfer between these two storage systems was to use an intermediate node to perform protocol translation. For this purpose, we designed three different data pipelines to transfer the data and to evaluate our system.

We had access to two cache nodes: one at SDSC (slic04.sdsc.edu) and other at NCSA (quest2.ncsa.uiuc.edu). The SRB and UniTree servers and the SRB cache node had gigabit Ethernet(1000 Mb/s) interface and the NCSA cache node had a fast Ethernet(100 Mb/s) interface. The local area network at SRB was a gigabit and the wide-area network was 622 Mbps ATM shared among all users. The bottleneck link was the fast Ethernet interface card on the NCSA cache node. Figure 3.11 shows the topology of the network, bottleneck bandwidth and latencies.

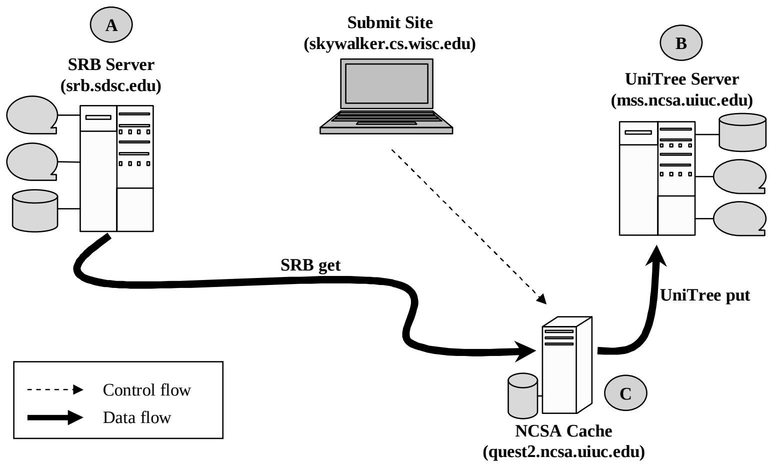


Figure 3.12: Data-pipeline with One Intermediate Node

First Data Pipeline Configuration

In the first data pipeline, we used the NCSA cache node, quest2.ncsa.uiuc.edu, to perform protocol translation. We transferred the DPOSS data from the SRB server to the NCSA cache node using the underlying SRB protocol and from the NCSA cache node to UniTree server using UniTree mssftp protocol. Figure 3.12 shows this pipeline configuration.

The NCSA cache node had only 12 GB of local disk space for our use and we could store only 10 image files in that space. This required careful space management and we had to remove a file immediately after transferring it to UniTree to create space for the transfer of next file.

We got an end-to-end transfer rate of 40Mb/s from the SRB server to the UniTree server. We calculated the end-to-end transfer rate by dividing the data transferred over a two-day period by the total time taken and repeating the process three times interleaved with other pipeline configuration in random order and averaging. We excluded the maintenance periods and other storage or network failures interrupting the transfers from the measurements. We also applied statistical methods and verified that day effect was statistically insignificant at 90-percentage level. We observed that the bottleneck was the transfers between the SRB

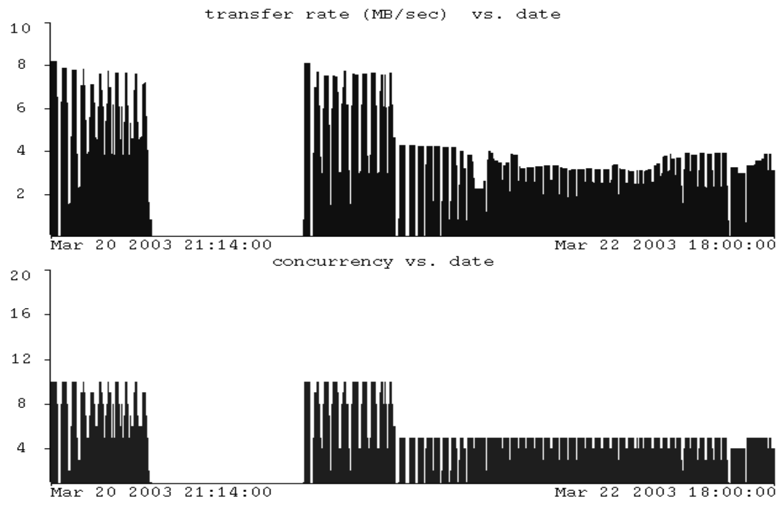


Figure 3.13: Pipeline 1: Throughput and Concurrency

server and the NCSA cache node. As SRB protocol did not allow us to tune TCP windows size forcing us to increase concurrency to achieve a similar affect, we decided to add another cache node at the SDSC site to regulate the wide area transfers.

Figure 3.13 is a snapshot showing the throughput and concurrency level of the system over time. There was a six-hour UniTree maintenance period during which the transfers stopped and then resumed. At some point, the SRB server started refusing new connections. The pipeline reduced the concurrency level automatically to decrease the load on the SRB server.

Second Data Pipeline Configuration

In the second pipeline configuration, we used both SDSC and NCSA cache nodes. We transferred the data from the SRB server to the SDSC cache node using the SRB protocol, then from the SDSC cache node to the NCSA cache node using third-party GridFTP transfers, and finally from the NCSA cache node to the UniTree server using UniTree mssftp protocol. Figure 3.14 shows this pipeline configuration. SDSC cache node also had space limitations requiring careful cleanup of transferred files at both cache nodes.

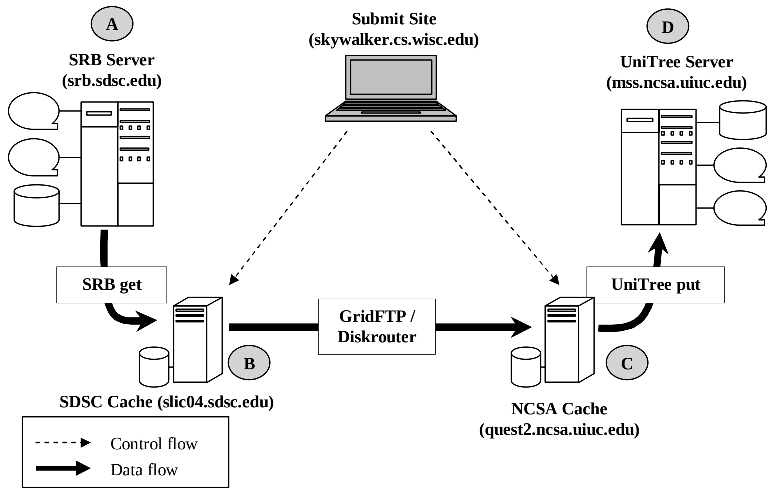


Figure 3.14: Data-pipeline with Two Intermediate Nodes

While this step may seem like an additional copy, we did not have source checksum. By transferring data to a local node, we were able to calculate the checksum and verify it at the destination. Using this configuration, we got an end-to-end transfer rate of 25.6 Mb/s, and the link between the SDSC cache node and the NCSA cache node was the bottleneck.

Figure 3.15 shows the throughput and concurrency level of the system over time. For optimal performance, we wanted a concurrency level of ten and the system maintained it. The fluctuations in the throughput are due to changing network conditions and GridFTP not aggressively utilizing the full bandwidth. We need to note here that we did not perform any tune-ups to the GridFTP transfers and used the default values for network parameters. Each transfer was performed using a single stream, but 10 concurrent transfers were performed at a time.

Third Data Pipeline Configuration

The third data pipeline configuration was almost the same as the second one, except that we replaced third-party GridFTP transfers between the SDSC cache node and the NCSA cache node, which were the bottleneck, with third-party DiskRouter [67] transfers.

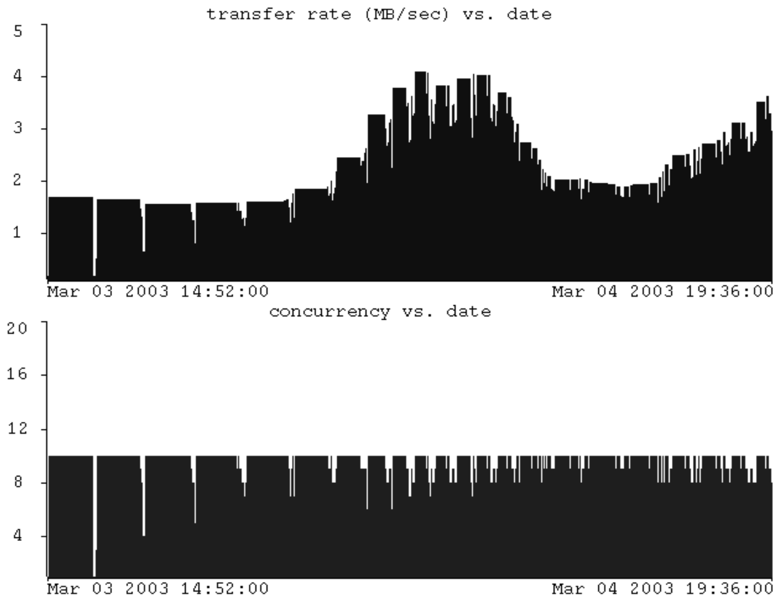


Figure 3.15: Pipeline 2: Throughput and Concurrency

This time we got an end-to-end throughput of 47.6 Mb/s. Disk-Router works best with a single transfer and it effectively utilizes the available bandwidth with a single transfer. Figure 3.16 shows two failures that occurred during the transfers. The first one was a UniTree server problem, and the second one was reconfiguration of DiskRouter that improved its performance. The system recovered automatically in both cases.

Comparison of Pipeline Configurations

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

Carefully tuning the I/O and socket buffer sizes in GridFTP would substantially improve

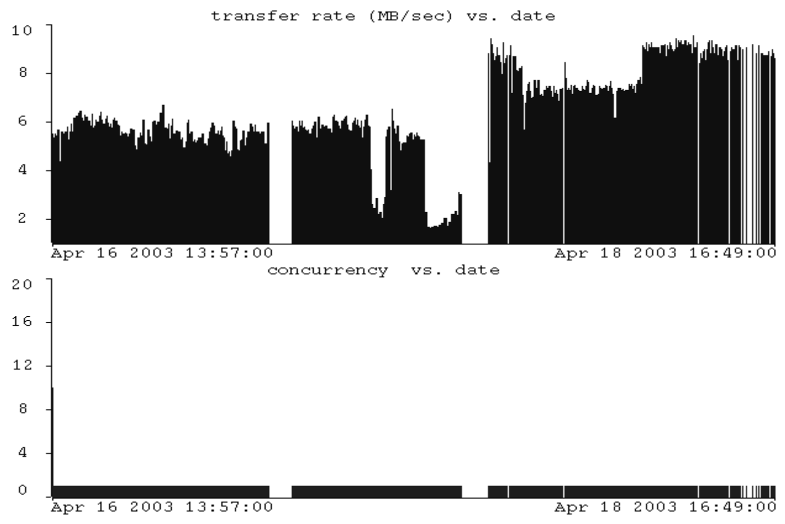


Figure 3.16: Pipeline 3: Throughput and Concurrency

Configuration	End to end rate
Pipeline 1	40.0 Mb/s
Pipeline 2	25.6 Mb/s
Pipeline 3	47.6 Mb/s

Table 3.1: End-to-end Performance of Different Pipeline Configurations.

its performance and we can integrate an agent that can do it dynamically into the data-pipeline. This also shows that running wide-area optimized protocols between cache nodes can improve performance enough to offset the penalty of an additional node.

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

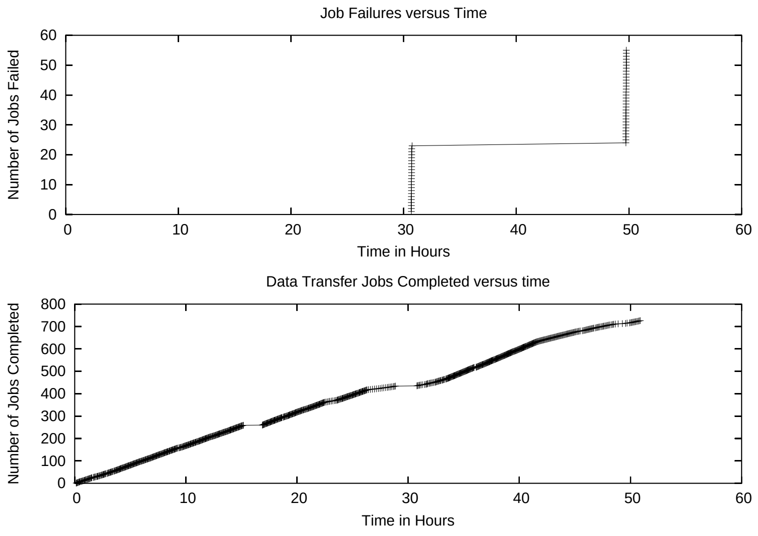


Figure 3.17: Pipeline 3: Automated Failure Recovery

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

Automated Failure Recovery

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

At times, either the source or the destination mass-storage systems stopped accepting new transfers. Such outages lasted about an hour on the average. In addition, we had windows of scheduled maintenance activity. We also had wide-area network outages, some lasting a couple of minutes and others lasting longer. While the pipeline was in operation, we had software upgrades. We also found a need to insert a timeout on the data transfers.

Occasionally we found that a data transfer command would hang. Most of the time, the problem occurred with third-party wide-area transfers. Occasionally, a third-party GridFTP transfer would hang. In the case of DiskRouter, we found that the actual transfer completed but DiskRouter did not notify us of the completion. Because of these problems, we set a timeout for the transfers. If any transfer does not complete within the timeout, Stork terminates it, performs the necessary cleanup and restarts the transfer.

Figure 3.17 shows how the third data pipeline configuration automatically recovers from two sets of failures. At around 15-17 hours, SRB transfers almost hung. They took a long time, around one hour and 40 minutes, but lesser than the two-hour time-out for a transfer. This could have been due to some maintenance or some other higher priority job using all the tape drives at the mass storage. The transfers did not fail but completed after that period, so it does not appear as failure. Around 30 hours, there was a short wide-area network outage. This resulted in DiskRouter failures. Another wide-area network outage at around 50 hours resulted in the second set of failures. The data pipeline recovered automatically from all these failures.

Figure 3.18 shows how the pipeline adapts the flow control on the fly. Around 4 hours, GridFTP encounters some wide-area failures and the pipeline lowers the number of concurrent transfers to seven. Close to 20 hours, SRB refuses new connection and the pipeline responds by trying to maintain a single connection. This affects the next hop and the number of concurrent GridFTP transfers drops to six. After that, UniTree accepts more connections and then slows down and this causes GridFTP to drop the number of concurrent transfers to five because of space limitations at the NCSA cache node. The next UniTree failure, at close to 100 hours, makes GridFTP drop the number of concurrent connections to four. The system was working through all these and users did not notice any failures. The end-to-end transfer rate observed by seeing the number of UniTree put transfers that completed show how well behaved the system is. Even though different flow control issues take place, the system is quite effective at maintaining the throughput.

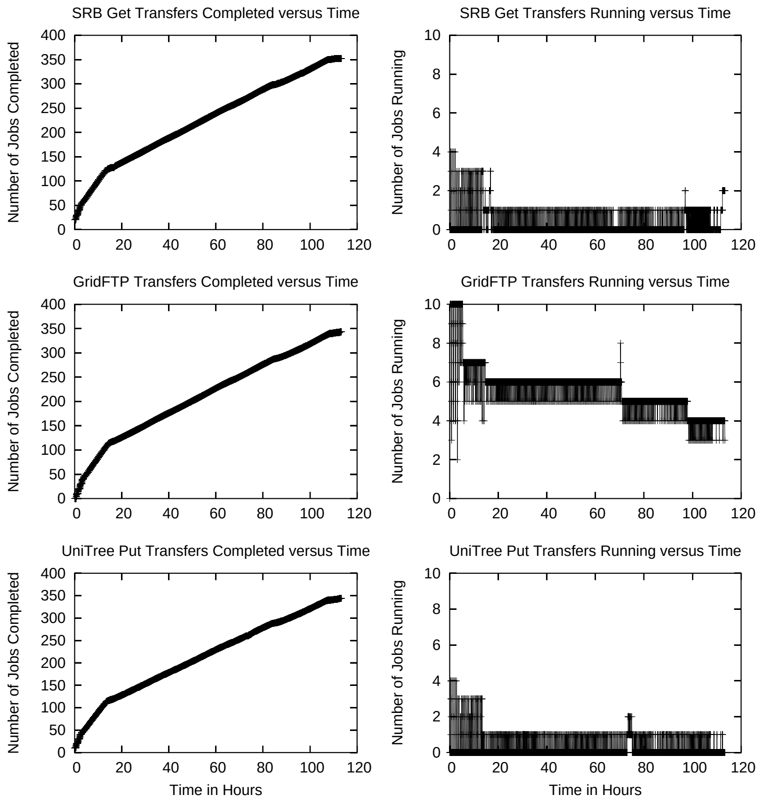


Figure 3.18: Pipeline 2: Flow control

3.4.2 WCER Educational Video Pipeline

Wisconsin Center for Educational Research(WCER) has nearly 500 terabytes of video on miniDV tapes and they want to make the original and MPEG1, MPEG2 and MPEG4 formats of the original electronically available at their storage server and SRB mass storage at San Diego supercomputing center(SDSC).

We helped them use our system to encode the videos to different formats and transfer them to SDSC. Since they provide researchers access to these videos via their *transana* [106] software, its meta-data showing the video locations has to be updated after a successful video transfer. The whole process is shown in Figure 3.19.

Because the Condor file-transfer mechanism used to move data from the stage-area to the individual nodes does not support files larger than 2 GB and most of the videos are

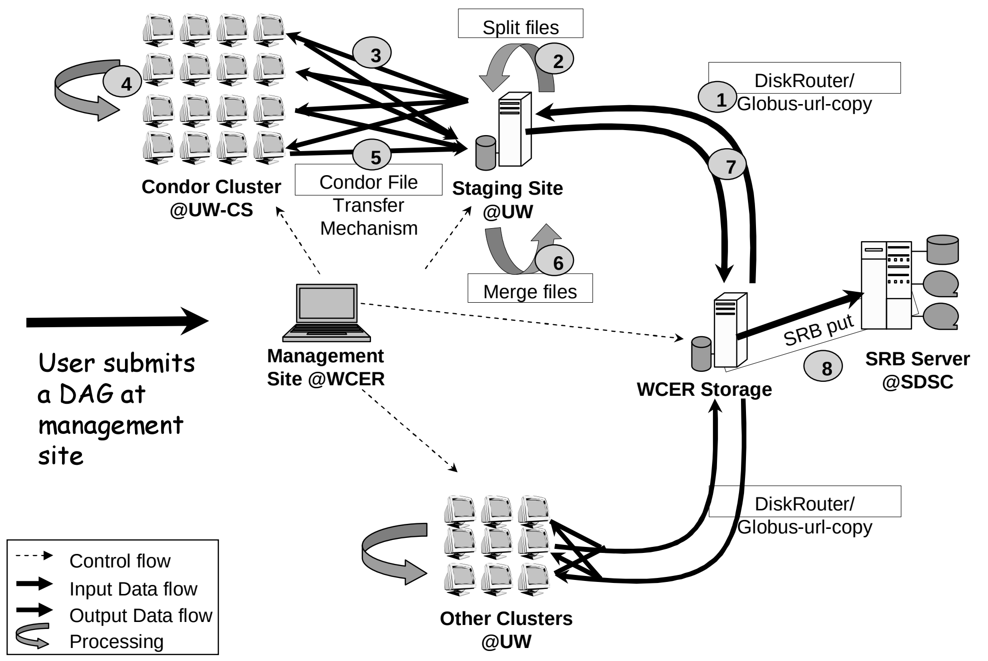


Figure 3.19: WCER Pipeline

around 13 GB, we split the video in the stage area and modify the processing job to merge the input files before processing.

The most suitable method for staging-in and staging-out the video depends on the cluster configuration and the characteristics of the computation such as the number of independent computation to be performed on the source video. In this section we present three designs and show the cases where each would be the most suitable.

Our system is not limited to a single cluster configuration and can make use of multiple heterogeneous clusters and any of the designs can be used in each. In each of the designs, we schedule the data movement taking into account network and end-host characteristics. By using network bandwidth and latency measurements, we tune the TCP buffer size to be equal to the bandwidth-delay product. Empirically, we found that as the number of concurrent transfers to/from a storage server increased, the throughput increased to a point and then started decreasing. Further, the number of concurrent transfers needed depended

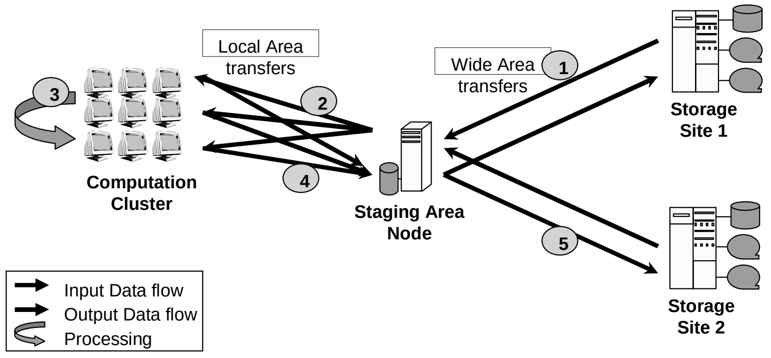


Figure 3.20: Designs 1 & 2 use Stage Area nodes of the Cluster

on the data-rate of the individual transfers. Using these empirical values observed during the course of the transfers, we tune the concurrency level to maximize throughput.

Design 1: Using Cluster Stage in/out Area

We allocate space on the stage area of the compute cluster, schedule the data transfer and after the completion of data transfer, we schedule the computation. This stage-in area is on the same local area network as the compute nodes. This configuration is shown in Figure 3.20

Certain clusters may have a network filesystem and depending on the data, users may want to use the network filesystem. If there is no network filesystem or the user prefers to access data from local disk (this case is preferred if the application does multiple passes through the video), then the data is moved from the stage-in area to the compute node using the mechanism provided by the computing system.

There may be multiple independent processing to be performed on the video, so the source video is deleted from the stage area and the space de-allocated only after all processing on this video have completed successfully. This also ensures that the data is staged only once from the remote storage thereby increasing performance by reducing wide-area data traversals.

Moving the data to the local area ensures that it takes deterministic time to move the data to the compute node. This bounds the amount of idle time the processor has to wait before performing the computation. Further, if the compute cluster has a network filesystem, we would be able to use it.

The stage-out process takes place in a similar fashion. If a network filesystem is not being used, space is allocated on the stage-area and the processed video is moved there and then it is scheduled to be transferred to the set of destinations. Once the processed video has been successfully transferred to all the destinations, it is deleted from the stage-area and the space is de-allocated.

Design 2: Optimizing the Stage in/out Process Using Hierarchical Buffering

Staging-in the data creates an unnecessary data-copy. We try to address this in the second design by using the hierarchical buffering. The hierarchical buffer server executes at the stage area nodes and tries to use memory and then disk to buffer incoming data. It creates logical blocks out of the data stream and performs management at that level. When sufficient blocks have been buffered to sustain transfer at local area network rates to the cluster node, a compute node is acquired and the hierarchical buffer server starts streaming incoming blocks and buffered blocks to the compute node.

If multiple independent computations are to be performed on the source video, the hierarchical buffer server sends a copy of the video to each of the compute node requiring that video thereby performing a multicast.

The hierarchical buffer client running on the compute node takes care of re-assembling the data into the file that the application wants.

This design is suitable for the case where we need to explicitly move the data to the compute node before execution. If the objective is to minimize the compute node idle time, the hierarchical buffer server can be made to call-back when enough data has been accumulated so that from now on it can transfer continuously at local area speeds to the

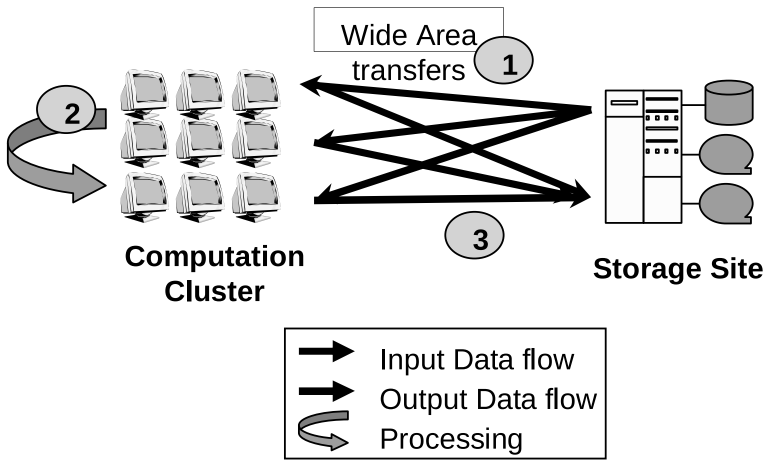


Figure 3.21: Design 3: Direct Transfer to Compute Nodes

compute node. If there are intermittent failures, the amount of data to be buffered before acquiring the compute node can be increased. Further, this amount can be dynamically tuned depending on the failure characteristics.

Design 3: Direct Staging to the Compute Node

In the third design as shown in Figure 3.21, we directly stage-in the data to the compute node. This requires that the compute nodes have wide-area access.

While acquiring the processor and starting the stage-in may waste processing cycles, this is more suitable if the scheduling system has compute on demand support. Here, the data transfer can take place at certain rate while the executing computation continues and when the data transfer is completed, the executing computation is suspended and the processor is acquired for the duration of our computation.

To acquire an processor, we need to have higher priority than the currently executing job. We need to do this carefully so that we pick idle nodes first and then randomly choose nodes that are executing lower priority jobs. The objective is to try to reduce starvation of certain job classes.

If there are multiple computation to be performed on the source video, we need to send the video to the other nodes as well. For this, we run the hierarchical buffer server on one computation node and make it write out a copy of the data to disk and stream the video to other compute nodes needing that video. This reduces the wide-area traversal of the source video to the minimum, but introduces more complexity.

Comparison of the Designs

The different designs are suitable for different conditions. The first design is simple and universal. The only requirement is the presence of a stage area. Most cluster systems provide that. It is suitable if a network filesystem is being used by the compute nodes. It also works if the compute nodes are in a private network and the stage-in area is the head-node with outside accessibility. It is the most robust and handles intermittent network failures well.

The second design gives a performance advantage over the first one if the data has to be explicitly staged to the compute node. We can use this design only if we have the ability to execute our hierarchical buffer server on the stage area nodes. Minimizing the disk traversal improves performance significantly.

If there a high failure rate or intermittent network disconnections, it may be better to use the first design instead of the second. Note, in this data is being streamed to the compute node after a certain threshold amount of data has been received. This does not do well, if failure occurs after start of streaming data to the compute node. This goes to the problem of finding a suitable threshold. Increasing the threshold improves the failure-case performance but decreases the normal case performance because it increases the amount of data that traverses the disk. At this point we dynamically set the threshold to be equal to the amount of data that has to be buffered to sustain transfer to the compute node at local-area transfer rates. This works well because more data is buffered for slow wide-area connection than for faster wide-area connections. This takes into account the failure characteristics as well

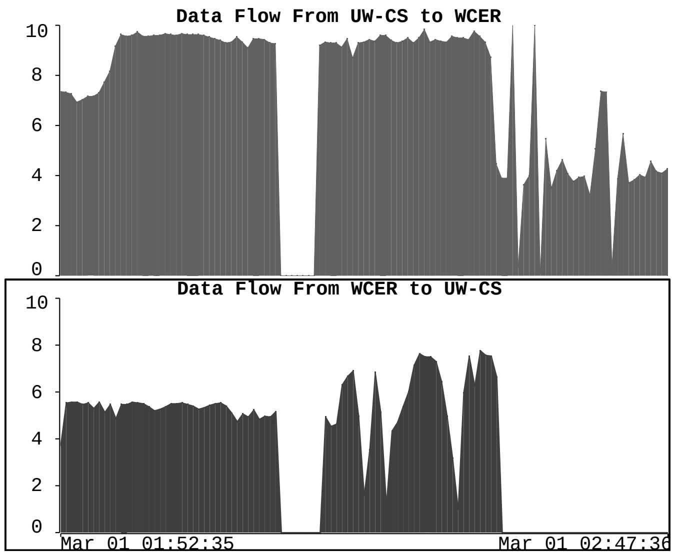


Figure 3.22: WCER Pipeline Data Flow

because failures and retries reduce the transfer rate requiring more data to be buffered before starting streaming.

The third design gives the best performance if the compute nodes have wide-area connectivity, computational scheduling system has a feature like compute-on-demand and the processor while executing has some extra cycles to spare for data transfer. The last condition happens when the processor has multi-threading support. Here, we need to be careful not to suspend other data-intensive jobs and also in acquiring processors so as not to starve certain job classes. Further, if other computation nodes need the data, we need to be run hierarchical buffer server to stream data to them. Failure handling is more complex in this case.

Results

We have first implemented the system using design 1, which transfers the video to the stage area of UW Madison computer science(UW-CS) cluster system using DiskRouter tools and then schedules the computation on that cluster. This is the configuration shown in

Figure 3.19.

We also tried using design 2 and design 3. Design 2 improved performance by about 20%, but design 3 slightly worsened the performance because we need to stream the data to two other nodes and in this cluster configuration, the stage-area node has gigabit connectivity while the compute nodes have only 100 Mbit connectivity.

Figure 3.22 shows the visualization of the data flow in design 1. The y-axis is in MBPS. We are performing tuning and get close to the maximum data transfer rate on the links. The link from UW-CS to WCER is limited by the 100 Mbit interface on the WCER machine while cross-traffic limits the data transfer rate from WCER to UW-CS.

The break in the middle is an artificial network outage we created. As it can be seen, the system recovers from the outage automatically without any human intervention. We also determined that we need two concurrent transfers for GridFTP and one transfer for DiskRouter to maximize the throughput of the storage system.

3.5 Discusssion

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. 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.

Chapter 4

Run-time Adaptation

4.1 Introduction

Sophisticated protocols developed for wide area data transfers like GridFTP [20] allow tuning depending on the environment to achieve the best performance. While tuning by itself is difficult, it is further complicated by the changing environment. The parameters which are optimal at the time of job submission, may no longer be optimal at the time of execution. The best time to tune the parameters is just before execution of the data placement job. Determining the environment characteristics and performing tuning for each job may impose a significant overhead. Ideally, we need an infrastructure that detects environmental changes and performs appropriate tuning and uses the tuned parameters for subsequent data placement jobs.

Many times, we have the ability to use different protocols for data transfers, with each having different network, CPU and disk characteristics. The new fast protocols do not work all the time. The main reason is the presence of bugs in the implementation of the new protocols. The more robust protocols work for most of the time but do not perform as well. This presents a dilemma to the users who submit data placement jobs to data placement schedulers. If they choose the fast protocol, some of their transfers may never complete and if they choose the slower protocol, their transfer would take a very long time. Ideally users would want to use the faster protocol when it works and switch to the slower more reliable protocol when the fast one fails. Unfortunately, when the fast protocol would fail is not known apriori. The decision on which protocol to use is best done just before starting the transfer.

Some users simply want data transferred and do not care about the protocol being used. Others have some preference such as: as fast as possible, as low a CPU load as possible, as minimal memory usage as possible. The machines where the jobs are being executed may have some characteristics which might favor some protocol. Further the machine characteristics may change over time due to hardware and software upgrades. Most users do not understand the performance characteristics of the different protocols and inevitably end up using a protocol that is known to work. In case of failures, they just wait for the failure to be fixed, even though other protocols may be working.

An ideal system is one that allows normal users to specify their preference and chooses the appropriate protocol based on their preference and machine characteristics. It should also switch to the next most appropriate protocol in case the current one stops working. It should also allow sophisticated users to specify the protocol to use and the alternate protocols in case of failure. Such a system would not only reduce the complexity of programming the data transfer but also provide superior failure recovery strategy. The system may also be able to improve performance because it can perform on-the-fly optimization.

In this chapter, we present the network monitoring and tuning infrastructure we have designed and implemented. The monitoring infrastructure determines the environment characteristics and detects any subsequent change. The environment characteristics are used by the tuning infrastructure to generate tuned parameters for the various protocols. These tuned parameters are fed to the data placement scheduler, Stork. Stork uses the tuned parameters while executing the data placement jobs submitted to it, essentially performing run-time adaptation of data placement jobs. Stork also has a dynamic protocol selection and alternate protocol fall-back capability. Dynamic protocol selection determines the protocols that are available on a particular host and uses an appropriate protocol for data transfer between any two hosts. Alternate protocol fall-back allows Stork to switch to a different protocol if the protocol being used for a transfer stops working.

4.2 Adaptation Methodology

The environment in which data placement jobs execute keeps changing all the time. The network bandwidth keeps fluctuating. The network route changes once in a while. The optic fiber may get upgraded increasing the bandwidth. New disks and raid-arrays may be added to the system. The monitoring and tuning infrastructure monitors the environment and tunes the different parameters accordingly. The data placement scheduler then uses these tuned parameters to intelligently schedule and execute the transfers. Figure 4.1 shows the components of the monitoring and tuning infrastructure and the interaction with the data placement scheduler.

4.2.1 Monitoring Infrastructure

The monitoring infrastructure monitors the disk, memory and network characteristics. The infrastructure takes into account that the disk and memory characteristics change less frequently and the network characteristics change more frequently. The disk and memory characteristics are measured once after the machine is started. If a new disk is added on the fly (hot-plugin), there is an option to inform the infrastructure to determine the characteristics of that disk. The network characteristics are measured periodically. The period is tunable. If the infrastructure finds that the network characteristics are constant for a certain number of measurements, it reduces the frequency of measurement till a specified minimum is reached. The objective of this is to keep the overhead of measurement as low as

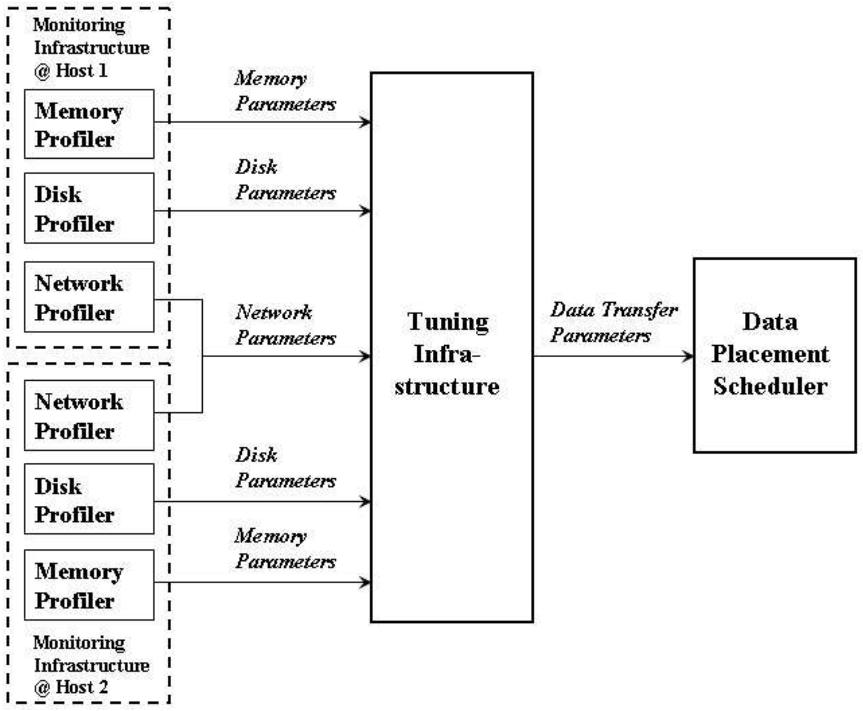


Figure 4.1: Overview of the Monitoring and Tuning Infrastructure

possible.

The disk and memory characteristics are determined by intrusive techniques, and the network characteristics are determined by a combination of intrusive and non-intrusive techniques. The memory characteristic of interest to us is the optimal memory block size to be used for memory-to-memory copy. The disk characteristics measured include the optimal read and write block sizes and the increment block size that can be added to the optimal value to get the same performance.

The network characteristics measured are the following: end-to-end bandwidth, end-to-end latency, number of hops, the latency of each hop and kernel TCP parameters. Since end-to-end measurement requires two hosts, this measurement is done between every pair of hosts that may transfer data between each other. The end-to-end bandwidth measurement uses both intrusive and non-intrusive techniques. The non-intrusive technique uses packet dispersion technique to measure the bandwidth. The intrusive technique performs actual transfers. First the non-intrusive technique is used and the bandwidth is determined. Then actual transfer is performed to measure the end-to-end bandwidth. If the numbers widely differ, the infrastructure performs a certain number of both of the network measurements and finds the correlation between the two. After this initial setup, a light-weight network profiler is run which uses only non-intrusive measuring technique. While we perform a longer initial measurement for higher accuracy, the subsequent periodic measurements are very light-weight and do not perturb the system.

4.2.2 Tuning Infrastructure

The tuning infrastructure uses the information collected by monitoring infrastructure and tries to determine the optimal I/O block size, TCP buffer size and the number of TCP streams for the data transfer from a given node X to a given node Y. The tuning infrastructure has the knowledge to perform protocol-specific tuning. For instance GridFTP takes as input only a single I/O block size, but the source and destination machines may have different optimal I/O block sizes. For such cases, the tuning finds the I/O block size which is optimal for both of them. The incremental block size measured by the disk profiler is used for this. The tuning infrastructure feeds the data transfer parameters to the data placement scheduler.

4.3 Implementation

We have developed a set of tools to determine disk, memory and network characteristics and using those values determine the optimal parameter values to be used for data transfers. We executed these tools in a certain order and fed the results to Stork data placement scheduler which then performed run-time adaptation of the wide-area data placement jobs submitted to it.

4.3.1 Disk and Memory Profilers

The disk profiler determines the optimal read and write block sizes and the increment that can be added to the optimal block size to get the same performance. A list of pathnames and the average file size is fed to the disk profiler. So, in a multi-disk system, the mount point of the different disks are passed to the disk profiler. In the case of a raid-array, the mount point of the raid array is specified. For each of the specified paths, the disk profiler finds the optimal read and write block size and the optimal increment that can be applied to these block sizes to get the same performance. It also lists the read and write disk bandwidths achieved by the optimal block sizes.

For determining the optimal write block size, the profiler creates a file in the specified path and writes the average file size of data in block-size chunks and flushes the data to disk at the end. It repeats the experiment for different block sizes and finds the optimal. For determining the read block size, it uses the same technique except that it flushes the kernel buffer cache to prevent cache effects before repeating the measurement for a different block size. Since normal kernels do not allow easy flushing of the kernel buffer cache, the micro-benchmark reads in a large dummy file of size greater than the buffer cache size essentially flushing it. The memory profiler finds the maximum memory-to-memory copy bandwidth and the block size to be used to achieve it.

4.3.2 Network Profiler

The network profiler gets the kernel TCP parameters from /proc. It runs Pathrate [44] between given pair of nodes and gets the estimated bottleneck bandwidth and the average round-trip time. It then runs traceroute between the nodes to determine the number of hops between the nodes and the hop-to-hop latency. The bandwidth estimated by Pathrate is verified by performing actual transfers by a data transfer tool developed as part of the

DiskRouter project [67]. If the two numbers differ widely, then a specified number of actual transfers and Pathrate bandwidth estimations are done to find the correlation between the two. Tools like Iperf [15] can also be used instead of the DiskRouter data transfer tool to perform the actual transfer. From experience, we found Pathrate to be the most reliable of all the network bandwidth estimation tools that use packet dispersion technique and we always found a correlation between the value returned by Pathrate and that observed by performing actual transfer. After the initial network profiling, we run a light-weight network profiler periodically. The light-weight profiler runs only Pathrate and traceroute.

4.3.3 Parameter Tuner

The parameter tuner gets the information generated by the different tools and finds the optimal value of the parameters to be used for data transfer from a node X to a node Y.

To determine the optimal number of streams to use, the parameter tuner uses a simple heuristic. It finds the number of hops between the two nodes that have a latency greater than 10 ms. For each such hop, it adds an extra stream. Finally, if there are multiple streams and the number of streams is odd, the parameter tuner rounds it to an even number by adding one. The reason for doing this is that some protocols do not work well with odd number of streams. The parameter tuner calculates the bandwidth-delay product and uses that as the TCP buffer size. If it finds that it has to use more than one stream, it divides the TCP buffer size by the number of streams. The reason for adding a stream for every 10 ms hop is that in a high-latency multi-hop network path, each of the hops may experience congestion independently. If a bulk data transfer using a single TCP stream occurs over such a high-latency multi-hop path, each congestion event would shrink the TCP window size by half. Since this is a high-latency path, it would take a long time for the window to grow, with the net result being that a single TCP stream would be unable to utilize the full available bandwidth. Having multiple streams reduces the bandwidth reduction of a single congestion event. Most probably only a single stream would be affected by the congestion event and halving the window size of that stream alone would be sufficient to eliminate congestion. The probability of independent congestion events occurring increases with the number of hops. Since only the high-latency hops have a significant impact because of the time taken to increase the window size, we added a stream for all high-latency hops and empirically found that hops with latency greater than 10 ms fell into the high-latency category. Note that we set the total TCP buffer size to be equal to the bandwidth delay product, so in steady state case with multiple streams, we would not be causing congestion.

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

The Parameter Tuner gets the different optimal values and generates overall optimal values. It makes sure that the disk I/O block size is at least equal to the TCP buffer size. For instance, the optimal disk block size may be 1024 KB and the increment value may be 512 KB (performance of optimal + increment is same as optimal) and the optimal TCP buffer size may be 1536KB. In this case, the parameter tuner will make the protocol use a disk block size of 1536 KB and a TCP buffer size of 1536 KB. This is a place where the increment value generated by the disk profiler is useful.

The Parameter Tuner understands different protocols and performs protocol specific tuning. For example, globus-url-copy, a tool used to move data between GridFTP servers, allows users to specify only a single disk block size. The read disk block size of the source machine may be different from the write disk block size of the destination machine. In this case, the parameter tuner understands this and chooses an optimal value that is optimal for both the machines.

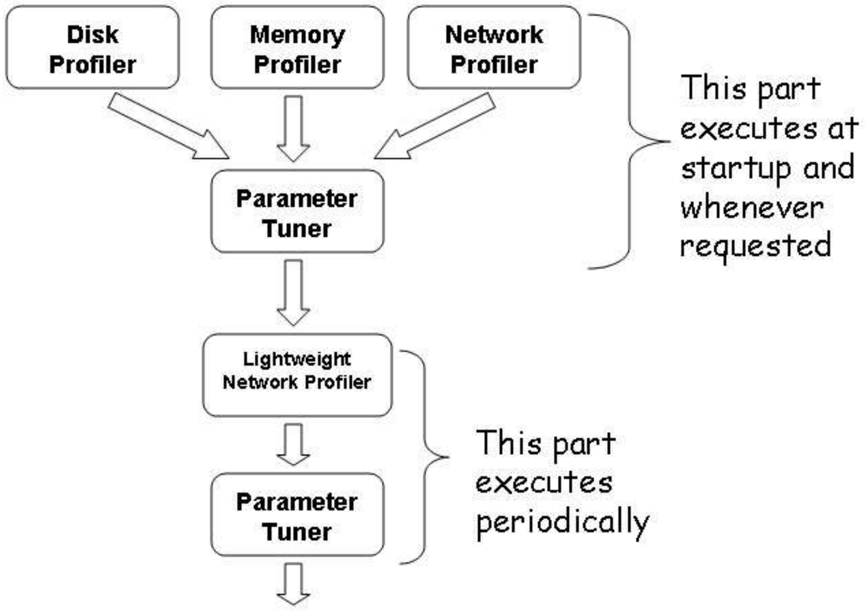

Disk Profiler
Memory Profiler
Network Profiler
Parameter Tuner
This part executes at startup and whenever requested
Lightweight Network Profiler
Parameter Tuner
This part executes periodically

Figure 4.2: Coordinating the Monitoring and Tuning infrastructure

4.3.4 Coordinating the Monitoring and Tuning Infrastructure

The disk, memory and network profilers need to be run once at startup and the lightweight network profiler needs to be run periodically. We may also want to re-run the other profilers in case a new disk is added or any other hardware or operating system kernel upgrade. We have used the Directed Acyclic Graph Manager (DAGMan) [4] [105] to coordinate the monitoring and tuning process. DAGMan is service for executing multiple jobs with dependencies between them. The monitoring tools are run as Condor [80] jobs on respective machines. Condor provides a job queuing mechanism and resource monitoring capabilities for computational jobs. It also allows the users to specify scheduling policies and enforce priorities.

We executed the Parameter Tuner on the management site. Since the Parameter Tuner is a Condor job, we can execute it anywhere we have a computation resource. It picks up the information generated by the monitoring tools using Condor and produces the different tuned parameter values for data transfer between each pair of nodes. For example if there are two nodes X and Y, then the parameter tuner generates two sets of parameters - one for transfer from node X to node Y and another for data transfer from node Y to node X. This information is fed to Stork which uses it to tune the parameters of data placement jobs submitted to it. The DAG coordinating the monitoring and tuning infrastructure is shown in Figure 4.2.

We can run an instance of parameter tuner for every pair of nodes or a certain number of pairs of nodes. For every pair of nodes, the data fed to the parameter tuner is in the order of hundreds of bytes. Since all tools are run as Condor jobs, depending on the number of nodes involved in the transfers, we can have a certain number of parameter tuners, and they can be executed wherever there is available cycles and this architecture is not centralized with respect to the parameter tuner. In our infrastructure, we can also have multiple data placement schedulers and have the parameters for data transfers handled by a particular scheduler fed to it. In a very large system, we would have multiple data placement schedulers with each handling data movement between a certain subset of nodes.

4.3.5 Dynamic Protocol Selection

We have enhanced the Stork scheduler so that it 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 in ClassAd [92] format which is a very flexible and extensible data model that can be used to represent arbitrary services and constraints.

```
[
  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. Stork detects a variety of protocol failures. In the simple case, connection establishment would fail and the tool would report an appropriate error code and Stork uses the error code to detect failure. In other case where there is a bug in protocol implementation, the tool may report success of a transfer, but stork would find that source and destination files have different sizes. If the same problem repeats, Stork switches to another protocol. The users also have the option to not specify any particular protocol 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 [12] 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 drop 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 the user's request. In case all the protocols fail, Stork will keep trying till one of them becomes available.

4.3.6 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 will bring to the system.

4.4 Experiments and Results

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.

4.4.1 Experiment 1: Testing the 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 4.3.

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. This was done to simulate a DiskRouter server crash. 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

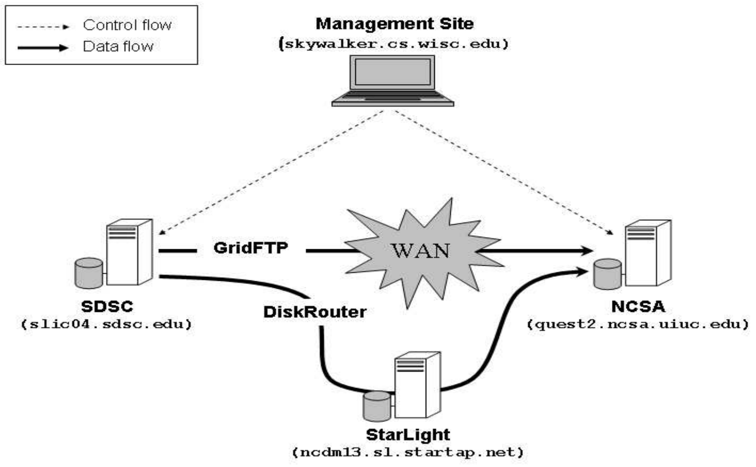


Figure 4.3: Experiment Setup

Figure 4.4. 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 immediately 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.

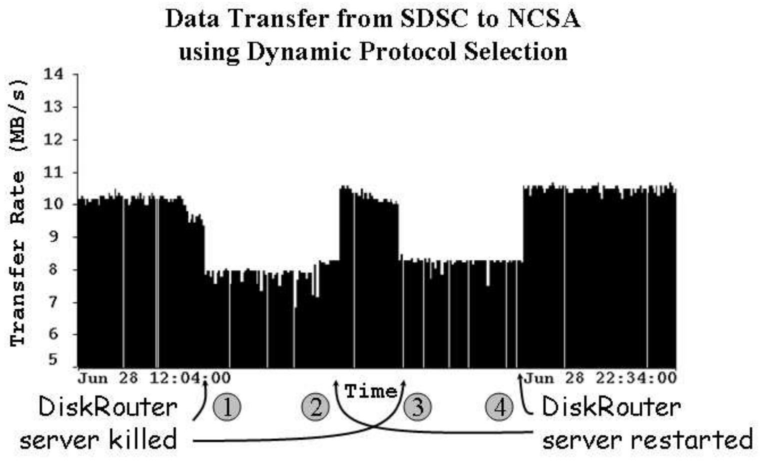


Figure 4.4: Dynamic Protocol Selection

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 4.1: Network Parameters for GridFTP Before and After Auto-tuning.

4.4.2 Experiment 2: Testing the Run-time Protocol Auto-tuning

In the second experiment, we submitted 500 data transfer requests to a traditional batch scheduler and 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.

The average data transfer rate that the traditional scheduler could get was only 0.5 MB/s. The default network parameters used by globus-url-copy are shown in Table 1. After a while, Stork started gradually tuning the network parameters. Every 15 minutes, Stork obtained the tuned-up values for I/O block size, TCP buffer size and the number of parallel TCP streams from the monitoring and tuning infrastructure. Then it applied these values to the subsequent transfers. Figure 4.5 shows the increase in the performance after Stork tunes up the network parameters. We got a speedup of close to 20 times compared to transfers with

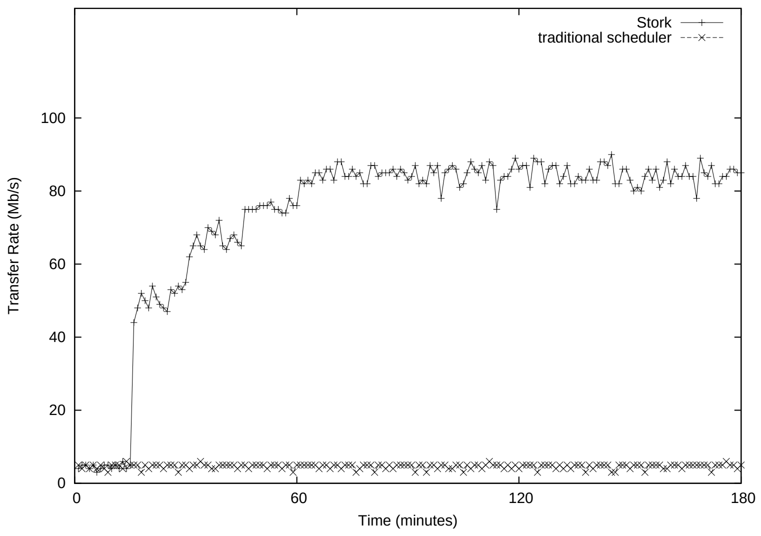


Figure 4.5: Run-time Protocol Auto-tuning

a traditional scheduler.

4.5 Discussion

In this chapter, we have shown a method to dynamically adapt data placement jobs to the environment at the execution time. We have designed and developed a set of disk and memory and network profiling, monitoring and tuning tools which can provide optimal values for I/O block size, TCP buffer size, and the number of TCP streams for data transfers. These values are generated dynamically and provided to the higher level data placement scheduler, which can use them in adapting the data transfers at run-time to existing environmental conditions. We also have provided dynamic protocol selection and alternate protocol fall-back capabilities to provide superior performance and fault tolerance. We have shown that our method can be easily applied and it generates better performance results by dynamically switching to alternative protocols in case of a failure, and by dynamically auto-tuning protocol parameters at run-time.

Chapter 5

Learning from History

5.1 Introduction

Grid computing [48] while enabling researchers to harness idle distributed resources, creates difficulties because of the lack of guarantees. Scaling an application from the controlled well-understood cluster environment to a Grid environment creates a plethora of problems.

Livny and Thain [82] [103] have pointed out that the only practical way of handling these problems is to make the client (submitting endpoint) responsible for the progress including failure handling. While client submit software like Condor [80] and Condor-G [52] address some of these problems, the presence of 'black holes', machines that accept jobs but never complete them, and machines with faulty hardware, buggy or misconfigured software impede the efficacy of using Grid based resources.

We study the prevalence of black holes by analyzing the log files of two real-life Grid applications. After detecting the presence of black holes, we investigate the reasons for their occurrence. As a viable solution, we introduce the concept of Grid knowledgebase that keeps track of the job performance and failure characteristics on different Grid resources as observed by the client.

Client middleware can use this knowledgebase transparently to improve performance and throughput of unmodified Grid applications. Grid knowledgebase enables easy extraction of useful information simplifying a variety of tasks including bug finding, statistics collection and visualization.

The mutually trusting entities can share the knowledgebase. Sites can use it to detect misconfiguration, software bugs and hardware faults. We discuss the design and implementation of our prototype Grid knowledgebase and evaluate its effectiveness on a real-life Grid workload.

5.2 Motivation

We performed an objective study to identify the presence of black holes by analyzing the log files of two real-life distributed applications: NCSA DPOSS image processing pipeline [43] and WCER video processing pipeline [63]. Both the pipelines strive towards a fully automated fault-tolerant processing of terabytes of images and videos respectively.

In the WCER pipeline log files, we found the presence of three black holes that accepted a job each and did not seem to have done anything and scheduler was trying unsuccessfully to talk to the machine for over 62 hours. We also found a case where a machine caused an error because of a corrupted FPU. Through a careful analysis, we found that certain machines had problems with certain job classes while they executed others successfully. As a particular case, we found that the machine that had FPU corruption with MPEG-4 encoding had earlier successfully performed MPEG-1 encoding.

Detecting the above kinds of problems is difficult and the only party affected is the job that was unable to complete successfully. Further, in a Grid environment, job submitter may not have control over the machine configuration. Following the 'dependability from client side' argument [82] [103], the job should be adapted to avoid those resources.

In an organization with thousands of compute nodes, it is a nightmare to ensure that all the different software are properly configured and working on all the nodes. While individual hardware and software are relatively easy to check, ensuring that different software work fine together is a non-trivial task. Most of the organizations at this point depend on user complaints to help them identify problems with such complex interactions. Many problems are not fixed because the users did not want to take the trouble of identifying the problem and reporting them.

In the WCER pipeline case, a couple of routine software and operating system upgrades fixed the problem. However, those upgrades took several months. The users did not try to find the cause of problem or report it because a few retries was probabilistically sufficient to get the jobs scheduled on a properly configured machine. A system capable of automatically identifying problem would greatly benefit site administrators. If site administrators use this information to fix the problems, it would result in better quality of service for jobs using that site.

To survive, organisms need to learn from experience and adapt themselves to changing environment. In a large-scale Grid environment, it is imperative that jobs should adapt to ensure successful completion. Just as organisms pass the gained wisdom down the generations, the wisdom gained from past jobs should be passed down to the future jobs.

We need a mechanism to enable passing this information from current jobs to future ones. To enable this, we propose the concept of Grid knowledgebase that aggregates the experience of the different jobs. It collects this information from the job log files produced by the batch scheduling systems like Condor/Condor-G and Stork. These log files are normally available to the client and are different from cluster/pool log files that many site administrators are unwilling to share. This log files essentially contain the view of the world as seen by the client.

We extract useful information from the log files and enter it into a database. We add an adaptation layer that uses this collected wisdom to adapt the failed job instances and future job instances of a distributed application. This is similar to organisms learning from experience and works well because many distributed applications consist of multiple instances of the same executable operating on different data.

5.3 Grid Knowledgebase Framework

Grid Knowledgebase, as shown in figure 5.1, consists of six main components: log parser, database, data miner, adaptation layer, notification layer, and visualization layer.

The log parser extracts useful information from the log files of submitted jobs and enters it into a database. This information includes the list of events that occurred during a job's life, the timestamp of each event, list of compute nodes that executed this job, resource utilization statistics and error messages.

The data miner runs a set of pre-selected queries on the database periodically. It checks the event history of the job and compares it with its own templates; checks the state of the resources on which the job executed; compares expected job execution time with the observed execution time; and tries to infer the reasons for delays, failures and other problems. In cases where the pre-selected set of queries is not sufficient, the user can either modify the existing queries or add new ones. The users can tune how often to run each of the queries and can even make them event-based so that certain events trigger execution of certain queries.

The data miner queries help determine the problem and choose an appropriate course of action. At present, the data miner runs three sets of queries.

The first set of queries takes a job-centric view and tries to find out the jobs that failed and tries to find the reason for them and feeds this to the adaptation layer.

The second set of queries takes a resource-centric view. They determine the resources that failed to successfully execute jobs and feed this information to the notification layer.

The third set of queries takes a user-centric view and tries to get the information that users may be interested in observing and feeds this to the visualization layer. An example

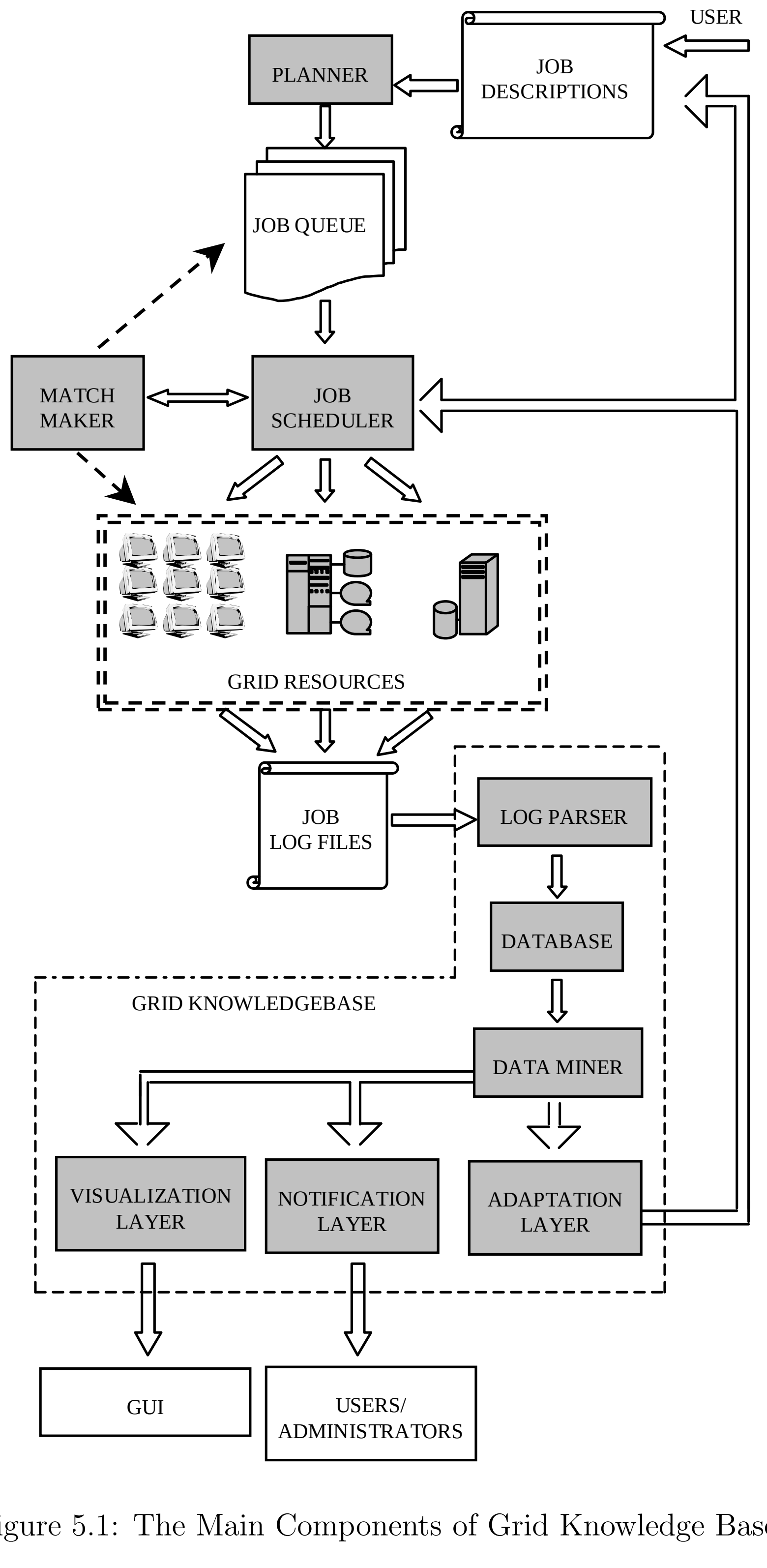


Figure 5.1: The Main Components of Grid Knowledge Base

would be tracking the memory usage of an application.

The adaptation layer analyzes the information fed by the data miner and if possible, tries to adapt the job dynamically to prevent recurrence of encountered problems. In certain cases, simple changes in job requirements such as increasing the memory requirement and/or hard disk space requirements or avoiding failure-prone machines may solve the problem. The adaptation layer can also pass this information and the strategy it took to higher-level planners like Chimera, Pegasus or DAGMan.

The notification layer informs the user who submitted the jobs and execute-site administrators about possible problems such as misconfigured and faulty machines. Since it is more important for machines bought under a project funding to be able to run that projects applications successfully than it is for those machines to run some other job when idle, the notification layer allows users/administrators to attach weights to machine-application pair. The email sent to the administrator specifies the failures and sorts them by weight. Users/administrators can tune the frequency of email notifications.

The visualization layer generates the information necessary to visualize the inferred information. Figure 5.1 shows how all of the components of Grid knowledgebase interact with each other and other entities in the overall scheduling system.

The different job classes using the framework can choose to share the data. Sharing this information between different organizations that submit the same classes of applications is very useful. In this case, the data miner can query remote data to get additional information. It is also possible to use a distributed database between different mutually trusted entities.

Organization may share this information even if they submit different application classes as the data miner can use the data to determine correlations between failures. For instance, a machine may fail when the input file is greater than 2 GB and we can use correlation to extract this information.

5.4 Implementation Insights

We have implemented a prototype of our grid knowledgebase framework and interfaced it with the Condor/Condor-G and Stork batch scheduling systems.

Condor-G uses Globus toolkit [49] functionality to schedule jobs on almost all grid-enabled resources. As Condor-G is being used by most of the Grid2003 [6] users to submit Grid jobs, the ability to parse Condor-G job logs gives us the ability to parse most of the real-life Grid job logs. Thus, most of the Grid community can easily use the grid knowledgebase and benefit from it.

After designing the parser, we had to choose a database. At first glance, we thought a native XML database would be a good choice for storing event data, since the log files were in XML format. As we could not find a free native XML database with suitable performance and because we found it difficult to construct XML (XQuery [36]/XPath [18]) queries to extract the useful information, we decided to load job logs into relational database tables. The current system uses a *postgres* [99] database.

We faced several issues while designing the schema to represent job event history. The first was whether we should use a vertical schema or a horizontal schema. In vertical schema, the events are stored as job id, event pairs. The horizontal schema allocates a field in the relational table for each of the events that may occur. Vertical schema is faster for loading, but requires joins for querying. Horizontal schema requires some processing before loading but is faster for querying, as it does not require any joins. Horizontal schema may waste space if the tuple is very sparse i.e. if most of the events rarely occur. After careful analysis, we found that the space wastage depended on the database implementation and that most relational databases are optimized for horizontal schemas. Further, vertical schemas required complex queries. With that, we decided to go in for a horizontal schema.

We now had to convert each of the events into a field of a relational table. We encountered

Field	**Type**
JobId	int
JobName	string
State	int
SubmitHost	string
SubmitTime	int
ExecuteHost	string []
ExecuteTime	string []
ImageSize	int[]
ImageSizeTime	integer []
EvictTime	int []
Checkpointed	bool []
EvictReason	string
TerminateTime	integer []
TotalLocalUsage	string
TotalRemoteUsage	string
TerminateMessage	string
ExceptionTime	int []
ExceptionMessage	string []

Table 5.1: Relational Database Schema Used to Store Job Event History.

the following problem. Certain jobs were executed multiple times because of encountered failures resulting in multiple event sequences for the same job. Further, even in a single event sequence, certain events like exception occurred multiple times.

Our initial approach was to create multiple table entries for such repeated events. We soon realized that querying them and extracting useful information was not straightforward. After some research, we found that *postgres* was an object-relational database and it supported 'array type', which essentially allows multiple entries in a single field. This addressed most of the issues we had. In the present form, each job maps into a single record in the database table.

Table 5.1 shows a simplified schema. To make it intuitive, we have simplified SQL varchar(n) to string and left out the bytes in the integer (we use int instead of int2, int4 and int8). The [] after a type makes it an array type. An array type can have a sequence of

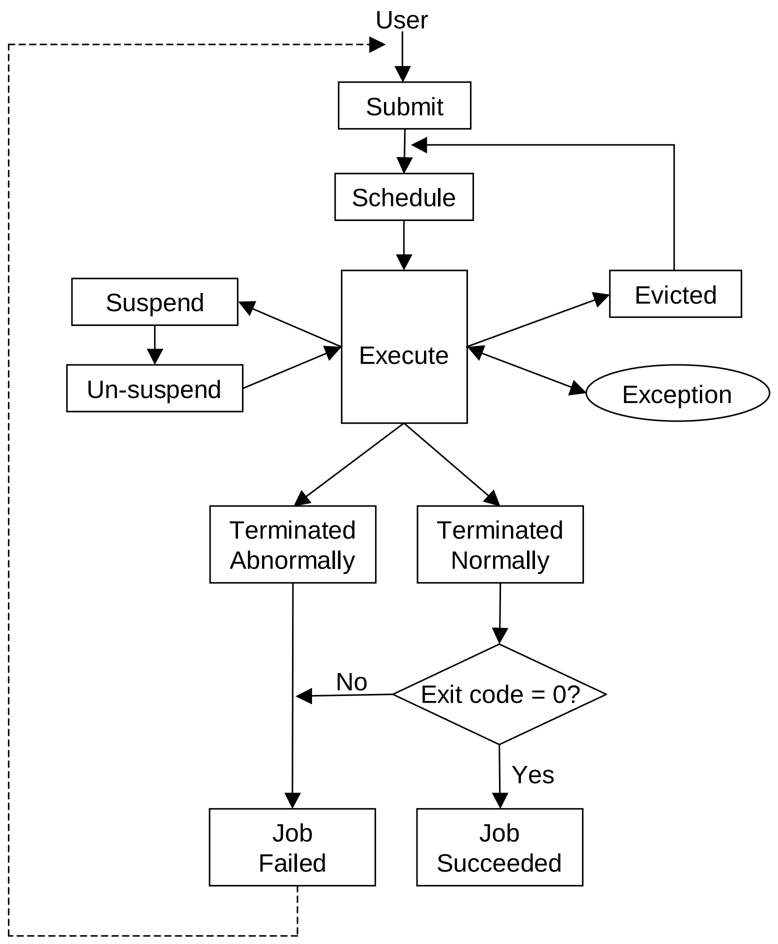


Figure 5.2: The State Changes a Job Goes Through.

values.

A job goes through different states in the course of its life and the state field tracks that. Figure 5.2 shows the state change that a job typically goes through. The job enters the system when a user submits it. When the scheduler finds a suitable host, it assigns the job to that host and starts executing it there. During execution, the scheduler may periodically observe the changes in the job state and log it. A number of exceptions may happen during job execution. For instance, the scheduler may be unable to talk to the execute machine because of a network outage.

An executing job may be evicted when the machine owner or a higher priority user wants to use the machine or when the job exceeds its resource usage limit. The evicted job is rescheduled. When a job is evicted, the job that has the ability to checkpoint may save its state and can resume from that state when it restarts. Finally, the job may terminate. If the job terminates abnormally, or it terminates normally with non-zero return value, the job is considered to have failed. If the job terminates normally with zero return value, it is considered successful.

The 'ExecuteHost' and 'ExecuteTime' are pairs in that first element of 'ExecuteTime' gives the time when the job started execution on the first 'ExecuteHost'. Other pairs are obvious from the first part of their field names.

5.5 Evaluation

Grid knowledgebase enabled us to extract useful information about jobs and resources and interpret them to gain a better understanding of failures. It helped us devise methods to avoid and recover from failures and helped us make better scheduling decisions. It helped us dynamically adapt our jobs to the ever-changing Grid environment. We observed that using the Grid knowledgebase in NCSA DPOSS image processing and WCER video processing pipelines increased their reliability and efficiency. Below, we list some of the contributions of Grid knowledgebase to these real life data processing pipelines.

Collecting Job Execution Time Statistics. We wanted to determine the average job execution time, its standard deviation, median, and fit a distribution to the execution time. Just the average and standard deviation is useful to benchmark two different clusters for this application. The Grid knowledgebase allowed us to easily extract this information.

A simplified query to extract the average and standard deviation of MPEG1 encoder is shown below.

```
SELECT AVG(TerminateTime[index]-ExecuteTime[index]),
```

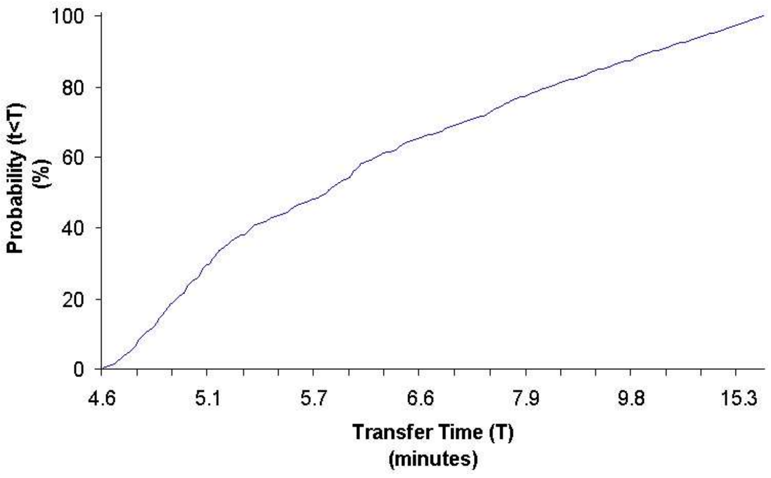


Figure 5.3: Catching Hanging Transfers

```
      STDDEV(TerminateTime[index]-ExecuteTime[index]),
  FROM  WCER_VideoPipeline
  WHERE TerminatedNormally[index] IS true
        AND JobName ILIKE('\%MPEG1-ENCODE\%') ;
```

Detecting and Avoiding Black Holes. During the processing of WCER video data, we detected the presence of some black holes. Jobs assigned to certain resources started execution but never completed. We called such machines black holes and decided to understand them and avoid them if possible.

To detect a black hole, we used the previously extracted run-time statistics. Our job execution times were normally distributed. So, we knew that 99.7% of the job execution times should lie between `average - 3*standard-deviation` and `average + 3*standard-deviation`.

Using the average and standard deviation calculated from Grid knowledgebase, we set the threshold to kill a job to `average + 3*standard-deviation`. If a job does not complete within that time, we marked that execution node as a black hole for our job and rescheduled the job to execute elsewhere.

Figure 5.3 shows the distribution of the more than 500 transfers we have performed between SDSC and NCSA. According to this information, we would expect 99.7% of the transfers to be completed in less than 16 minutes. If any transfer takes more than 16 minutes, the scheduler would suspect that there is something going wrong with the transfer. It would kill the current transfer process and restart the transfer. Of course we can always setup the threshold higher to prevent false positives.

The standard deviation takes the performance difference between Grid resources into account. If we want to detect sooner at the expense of false positives, we can decrease the threshold to `average + 2*standard-deviation`. Even with this threshold, we would only be rejecting around 4% of the machines, these would be the top 4% of slow machines, and this selective reject may considerably improve the throughput. Users can tweak this mechanism to improve throughput by avoiding a certain top fraction of the slow machines.

It is also possible to parameterize this threshold taking into account factors like input file size, machine characteristics and other factors. For this, we need to use regression techniques to estimate the mean and standard deviation of job-run-time for a certain input file size and machine characteristics. This parameterization would enable generation of tighter estimates and quicker detection of black holes.

5.6 Other Contributions

Grid knowledgebase has other useful contributions that we did not use during the execution of our two real life data processing pipelines but would be useful to the Grid community. We list some of them below.

Identifying Misconfigured Machines. We can easily identify machines where our jobs failed to run. In our and other environments, we find that it is important for machines bought under a particular project grant to be able to successfully execute job classes belonging to that project. We also care about failures of other job classes that use our idle CPUs but they are not that important. To address this, we attach a weight to each of the different job classes. Extracting the failures observed by the different job classes and multiplying by the associated weight and sorting, we can get the list of machines ordered by priority that site administrator needs to look into.

We can also extract additional information by using correlation to help site administrator find the fault. For example, we can find that a certain set of machine fails for jobs that have an input file size that is larger than 2 GB. Site administrators can use this information to find the exact problem and fix it.

Identifying Factors affecting Job Run-Time. Some users may want to add more computing power to enable additional processing. They need to determine the suitable machine configuration. Using our system, they can extract the run-time on the different resource and try to extract the significant factors affecting the job performance. This would help them choose appropriate machine configuration.

Bug Hunting. Writing bug-free programs is difficult. Since the scheduler logs the image-size of the program and as the log-parser enters this into the database, the data miner can query this information and pass it to the visualization layer to graph the growth in memory size. A continuously growth may indicate the presence of a memory leak. Grid application developers can use this information to identify bugs. It is very difficult to find bugs that occur only on certain inputs. A simple query can find out the inputs that caused a high growth in memory size. Similarly, if a job fails on certain inputs, the Grid knowledgebase can automatically derive this and email this information to the application developer.

Figure 5.4 shows Grid knowledgebase finding a memory-leaking process. Condor pre-empted and rescheduled the job three time before the Grid knowledgebase categorized it as a memory leaking process and notified the job submitter. The submitter fixed the memory leaking code and resubmitted the job. After resubmission, we see that the image size of the job stabilized at a certain point and does not increase any more.

Application Optimization. Many grid applications contain a number of different

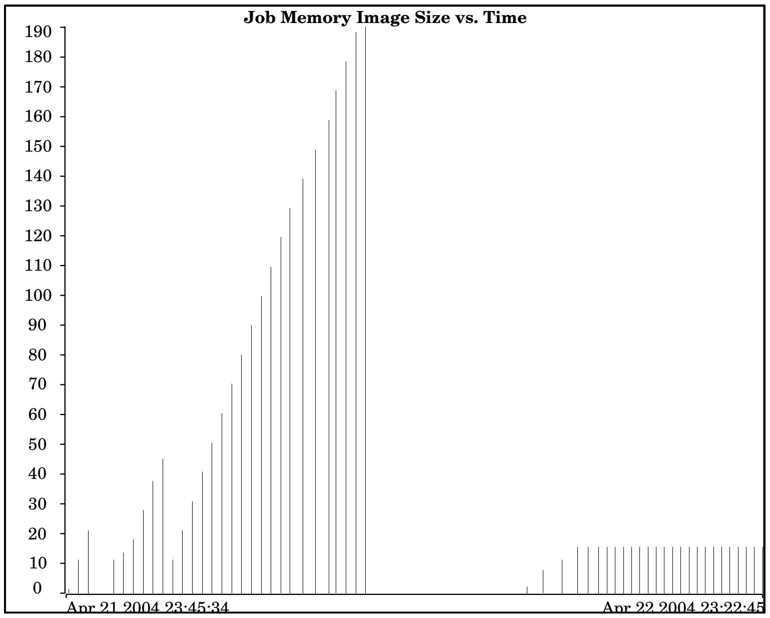


Figure 5.4: Catching a Memory Leak

processing steps executed in parallel. Grid application developers want to parallelize the steps depending on the time each takes to execute. While they try to do this by executing on a single machine, they would really like feedback on what happens in a real Grid environment. At present, they find it difficult to extract this information. With the Grid knowledgebase, application developer can write a simple two-line SQL query to extract this information and use it to redesign the application.

Adaptation. Smart Grid application deployers can come up with easy triggers to adapt the jobs to improve throughput. A simple example is to find the machines that take the least time to execute this job class and try to preferentially choose them over other machines if those machines are available.

Extensibility. The log parser parses XML logs. This is very extensible and users can even extend this mechanism to parse the logs produced by application itself. This may yield useful information.

5.7 Discussion

We have introduced the concept of Grid knowledgebase that keeps track of the job performance and failure characteristics on different resources as observed by the client side. We presented the design and implementation of our prototype Grid knowledgebase and evaluated its effectiveness on two real life distributed data processing pipelines. Grid knowledgebase helped us classify and characterize jobs by collecting job execution time statistics. It also enabled us to easily detect and avoid black holes.

Grid knowledgebase has a much wider application area and we believe it will be very useful to the whole distributed systems community. It helps users identify misconfigured or faulty machines and aids in tracking buggy applications.

Chapter 6

Failure Detection and Classification

6.1 Introduction

A major hurdle facing data intensive applications is appropriate handling of failures that occur in an opportunistic environment. Most application developers are unaware of the different types of failures that may occur in such an environment. Understanding and handling failures imposes an undue burden on the application developer already burdened with the development of their complex distributed application.

We feel that the middleware should tolerate faults and make this functionality transparent to the application. This would enable different data intensive applications to become fault-tolerant without each having to pay a separate cost. Removing the burden of understanding and handling failures lets application developers concentrate on the problem at hand and reduces the time to the solution for their problem.

We have developed a transparent middleware solution, Phoenix [64], that adds fault-tolerance to data intensive application by detecting failures early, classifying failures into transient and permanent, and handling each transient failure appropriately.

6.2 Background

In this section, we give the widely accepted definitions for faults, errors and failures, and discuss the issue of error propagation.

Faults, Errors and Failures

The widely accepted definition, given by Avizienis and Laprie [25], is as follows. A fault is a violation of a system's underlying assumptions. An error is an internal data state that reflects a fault. A failure is an externally visible deviation from specifications.

A fault need not result in an error nor an error in a failure. An alpha particle corrupting an unused area of memory is an example of a fault that does not result in an error. In the Ethernet link layer of the network stack, a packet collision is an error that does not result in a failure because the Ethernet layer handles it transparently.

Error Propagation

In a multi-layered distributed system where layers are developed autonomously, what errors to propagate and what errors to handle at each level is not well understood [102]. The end-to-end argument [94] states that the right place for a functionality is the end-point, but that it may be additionally placed in the lower levels for performance reasons.

Pushing all the functionality to the end-point increases its complexity and requires the end-point developers to understand all errors that may occur in the underlying layers. In an opportunistic computing environment, where application developers are domain experts and not necessarily distributed computing experts, requiring application developers to understand all different types of errors would mean that they might never complete their application.

An alternate approach followed in many multi-layered systems including the network stack is to make each layer handle whatever error it can and pass up the rest. This masking of errors, while reducing higher-level complexity, hurts the performance of sophisticated higher layers that can use this error information to adapt.

Thain and Livny [102] have developed a theory of error propagation. They define error scope as the portion of the system an error invalidates and state that an error must be propagated to the program that manages its scope. Applying their theory, we find that the errors in an opportunistic widely distributed environment are of middleware scope and not necessarily of application scope. Therefore, the middleware layer may handle most error types. Aided by this theory, we decided to add fault-tolerance capability at the middleware level. To handle the information loss and to enable sophisticated applications and allow interposition of adaptation layers between our middleware layer and the application, we persistently log the errors encountered and allow tuning of the error masking. Logging of the errors helps performance tuning and optimization.

6.3 Faults and Failures in Distributed Systems

We analyzed the faults and failures in large distributed systems by looking at two large distributed applications: US-CMS and BMRB BLAST, each of which was processing terabytes of data and using hundreds of thousands of CPU hours. We also analyzed several other small applications running in the Condor pool at UW-Madison campus having a couple of thousand compute nodes. The failures we have observed [66] are:

Hanging Processes. Some of the processes hang indefinitely and never return. From the submitters point of view there was no easy way of determining whether the process was making any progress or was hung for good. The most common cause of hanging data transfers was the loss of acknowledgment during third party file transfers. In BMRB BLAST, a small fraction of the processing hung and after spending a large amount of time, the operator tracked it down to an unknown problem involving the NFS server where an NFS operation would hang.

Misleading Return Values. An application returning erroneous return values is a very troublesome bug that we encountered. We found that even though an operation failed, the application returned success. This happened during some wide area transfers using a widely used data transfer protocol. We found that if the destination disk ran out of space during a transfer, a bug in the data transfer protocol caused the file transfer server to return success even though the transfer failed. This in turn resulted in failure of computational tasks dependent on these files.

Data Corruption. Faulty hardware in data storage, staging and compute nodes corrupted several data bits occasionally. The faults causing this problem included a bug in the raid controller firmware on the storage server, a defective PCI riser card, and a memory corruption. The main problem here was that the problem developed over a course of time, so initial hardware testing was not effective in finding the problems. The raid controller firmware bug corrupted data only after a certain amount of data was stored on the storage server and hence was not detected immediately after installation. In almost all of these cases, the data corruption happened silently without any indication from hardware/operating system that something was wrong. Tracking down the faulty component took weeks of system administrator's time on average.

Hanging Processes. Some of the processes hang indefinitely and never return. From the submitters point of view there was no easy way of determining whether the process was making any progress or was hung for good. The most common cause of hanging data transfers was the loss of acknowledgment during third party file transfers. In BMRB BLAST, a small fraction of the processing hung and after spending a large amount of time, the operator tracked it down to an unknown problem involving the NFS server where an NFS operation would hang.

Misbehaving Machines. Due to misconfigured hardware or buggy software, some machines occasionally behaved unexpectedly and acted as 'black holes'. We observed some computational nodes accepting jobs but never completing them and some completing the job but not returning the completion status. Some nodes successfully processed certain job classed but experienced failures with other classes. As a particular case, in WCER video processing pipeline [63], we found that a machine that had a corrupted FPU was failing MPEG-4 encoding whereas it was successfully completed MPEG-1 and MPEG-2 encodings.

Hardware/Software/Network Outages. Intermittent wide area network outages, outages caused by server/client machine crashes and downtime for hardware/software upgrades and bug fixes caused failure of the jobs that happened to use that feature during that time.

Over commitment of Resources. We encountered cases where the storage server crashed because of too many concurrent write transfers. We also encountered data transfer time-outs that that were caused by storage server trashing due to too many concurrent read data transfers.

Insufficient Disk Space. Running out of disk space during data stage-in and writing output data to disk caused temporary failures of all involved computational jobs.

6.4 Detecting Failures

Complexity of distributed systems makes failure detection difficult. There are multiple layers from the hardware to the application. The end-to-end argument [94] states that the end-point is the right place for a functionality. In this case, the end-point is the application itself. This application level failure detection through check summing and verifying data has been done in Google file system [54], proving this approach to be feasible. Since we did not want to impose an undue burden on application developers to handle failure detection

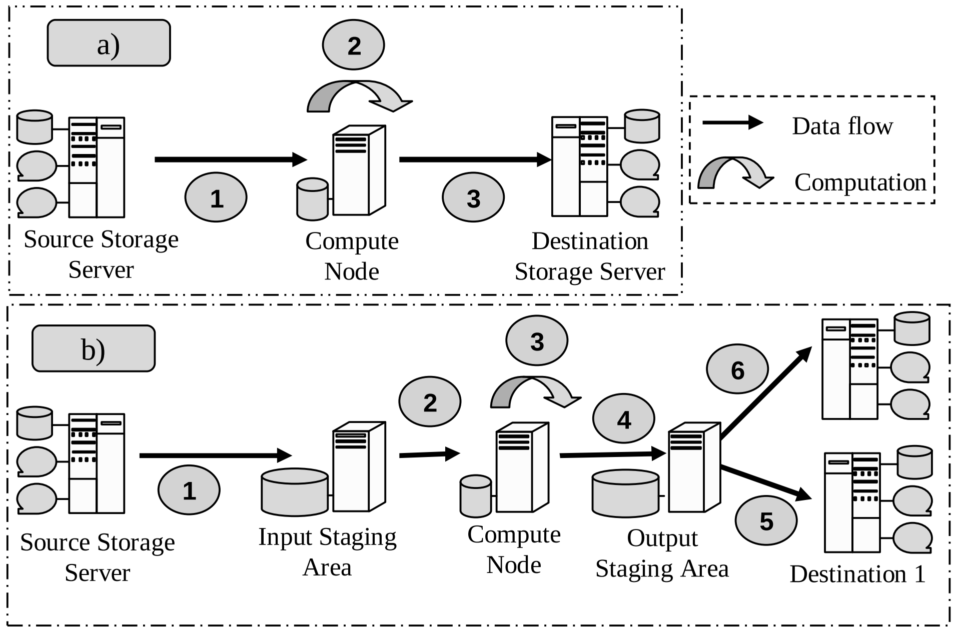


Figure 6.1: Stages in Distributed Processing

and handling, we implemented the error/failure detection on top of the application itself, by verifying that results generated are correct.

In addition to detecting erroneous results, we also need to detect the cause of the fault and possibly replace that faulty component. This is also in accord with the end-to-end argument that states that performance reasons may motivate a functionality implementation at lower layers in addition to the end-points. Identifying the source of the erroneous result has so far been a *'black art'* in the realm of select few system administrators and operators. This process takes considerable amount of time, usually weeks, expending considerable amount of human resources.

Figure 6.1 shows all components involved in a typical processing. As shown in figure 6.1 a), a typical processing consists of moving the source data from source storage server to the compute node, performing computation on the compute node and then transfer the result to the destination storage server. There could be multiple compute nodes if it is a parallel job. Figure 6.1 b) shows a scenario that includes intermediate stage nodes. Input stage area could

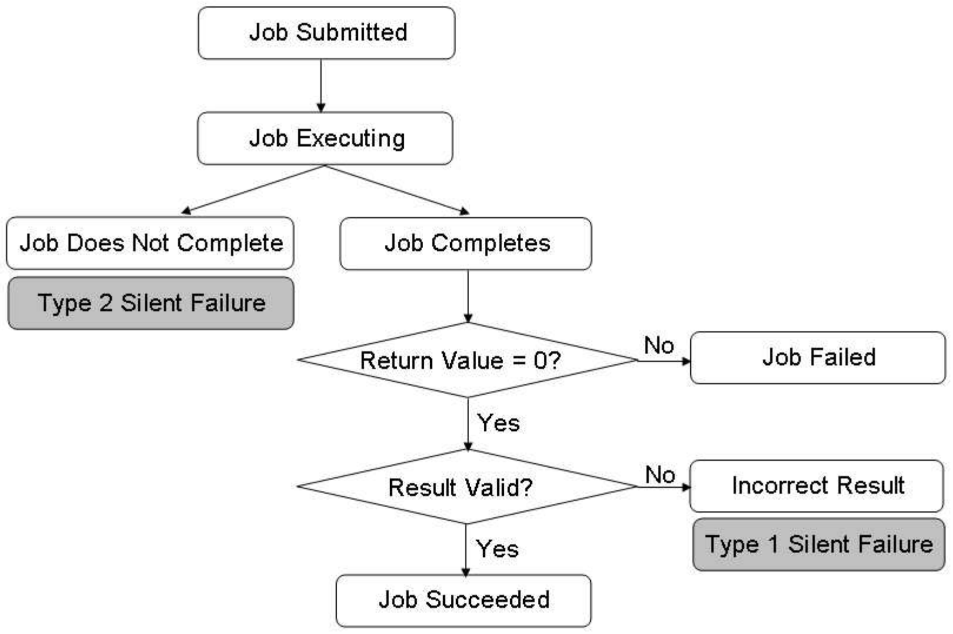


Figure 6.2: Type I and Type II Silent Failures.

be used for any of the following reasons: data resides in tapes on a mass storage system, to cache data that may be used again and source storage server is at a remote location. A destination stage area may be used for caching, data distribution to multiple destinations and to enable the data transfer to a remote storage server to be scheduled. In [69], we list in detail the cases where such intermediate nodes are needed, useful and give some real world examples of systems using such intermediate stage nodes.

We classify silent failures into two types as shown in figure 6.2. Type I silent failures are silent failures that give incorrect results without any error status indication. Type II silent failures are silent failures in which the process or transfer just hangs. Type I gives a successful return code and shows that the process is completed but the results are incorrect. Type 2 never returns, so user cannot find out if the process will complete. This could be at times due to bugs. In addition to silent failure, jobs may fail with an error status and they are easier to detect. We will first discuss about handling Type I.

We want to detect if a failure has occurred and if we need to track down the cause of that failure. Normally, a silent failure of lower level component may result in a failure higher up the chain and to track down the fault, we may need to go down the hierarchy. For instance, the cause of a computation failure may be because of data corruption in the intermediate storage server and this in turn may be caused by a faulty RAID controller in the storage server. We feel that automatically isolating the fault to whole system boundary is easier and this would aid the system administrator in locating the exact problem.

Consider a simple case where the user has to be 100% certain that the result is correct. A simple way of doing that is to compute the result twice and verify that they match. While doing this we need to be careful to ensure that the two computations do not overwrite the same data. Name space mapping can address this. Suppose if we find that a result is incorrect, we can pick up all the incorrect results in a given time period and all systems interacted with most of the results is the likely culprit. A simple mechanism that detects this can notify it to the system administrator who can then test that system. At the same time, the component can give feedback to higher-level planners like Pegasus [40] and/or distributed schedulers to ensure that they do not use this resource until the fault has been resolved. Verification of data transfers involves checksum generation and verifying that source and destination checksums match.

The components in the system should be tested to determine a failure. The methodology for testing can be inferred from “THE“ multiprogramming system [42], where they had a layered structure to test that reduced the number of test cases. We believed that a conscientious distributed system designer should design such a test infrastructure. We are aware that the NSF sponsored National Middleware Initiative (NMI) [10] is in the process of designing and implementing such a test infrastructure. If such a test infrastructure exists, the mechanism on detecting a failure can trigger a test of the whole system to isolate the faulty component. As an alternative, to isolate machine faults at a coarse grain, a tester can periodically execute a test program that generates a known result and takes a certain

Application	**Coefficient of Variation**
BLAST BMRB (1MB Database)	0.19
BLAST PDB (69MB Database)	0.34
BLAST NR (2GB Database)	0.49
NCSA Sextractor Processing	2.00
NCSA Data Transfer	1.00

Table 6.1: Coefficient of Variation

deterministic amount of time on each machine. If any machine gives a wrong result or the run time deviates considerably, the system administrator can be informed of the problem.

If the user does not want to pay a 100% overhead by performing each computation twice and if a testing system exists, he can specify the fraction of extra computation that he is willing to perform. The failure detector will inject that fraction of extra computation into the distributed system in a statistically unbiased manner. The results of these extra computations are compared with results of the previous execution and verified to be the same. In case of difference, the failure detector can tag those machines and perform the computation again on a different machine to identify the faulty one. When the failure detector identifies a faulty machine, it can report the time from the successful machine test to current time as time when the machine was in a possibly faulty state. Results generated using that machine during that time may have to be recomputed.

Handling Type II silent failures requires some more effort. The issue is whether it is possible to detect such a failure. In practice, most of the hung processes have a way of detecting that they have failed to make forward progress. A simple case is that of data transfer, we can find out how the file size varies over time and if the file size does not change for a long period, we can know that the file transfer has hung. Another way is to come up with reasonable time-outs for operations. We can find out that a transfer or computation has hung if it does not complete in a certain period.

Most of the present day distributed workloads consist of a large number of instances of the

Source URL (protocol://host:port/file)	**Destination URL**	**Error**
gsiftp://quest2.ncsa.uiuc.edu:4050/tmp/1.dat	nest://beak.cs.wisc.edu/tmp/1.dat	1

Table 6.2: Information Fed to the Failure Agent.

same application. Typically, the standard deviation across is of the same order of magnitude as mean if not lesser. This lends a very effective way to detecting Type II failures. To handle Type II failures, we use our Grid Knowledgebase [62] and extract the history of transfers/computation and fit a distribution to the transfer/compute times. Depending on the user specified threshold of false positives(e.g. 1%, 5%, etc), we set the time-limit for a transfer/computation to be mean + x(standard-deviation), where x is derived from the false-positive threshold and the distribution fitted. If no history exists, we do not set any threshold for the computation, but set a threshold for the data transfer based on a minimal data transfer rate. If the transfer/computation exceeds the threshold, the failure-detector stops it and marks it as a failure.

Users can specify policy on what fraction of the processing they are willing to re-do. If users want responsiveness, they may trade some extra processing and set a lower threshold. If they want to minimize the overhead, they would use a higher threshold.

To evaluate our ability to identify silent Type II failures, we looked at the co-efficient of variation of some applications. Table 6.1 shows the co-efficient of variation of a few well known applications. We found that the coefficient of variation of all classes we encountered were less than four, which shows the applicability of our method.

6.5 Classifying Failures

After identifying failures, we need to classify them. In the computation side, researchers have done most of the work to differentiate between resource failure and application failure. Most interfaces report if the job failure was due to middleware/resource error [52]. New problems encountered when jobs run inside a virtual machine like Java Virtual Machine have been handled by having a wrapper around the job to correctly propagate the error [102].

On the data placement side, the issue of separating transient errors from permanent errors has not been addressed. It is made difficult by the lack of feedback from the underlying system and at times, even the underlying system may not know how to classify the error. For instance, if the source host is unreachable, either it may mean there is a temporary network outage (transient error) or that there was a typo in source host in the data transfer job (permanent error).

To classify data placement failures, we propose a failure agent that takes the complete source and destination URL and optionally the return code, as shown in Table 6.2, and identifies the cause of the failure and interacts with the policy manager and classifies the failure into transient or permanent and gives a detailed error status.

Failure Agent

As shown in figure 6.3, the failure agent identifies the source of failure as follows. The failure agent checks if the appropriate DNS server is up.

Next, the failure agents checks if the source and destination have a valid DNS entry. If any does not, it is likely that the user made a typo.

If the DNS entry exists, it tries to see if that host network is reachable. If that network is not accessible, it logs it as wide-area network failure. As the failure agent may be running on a different network from the source or destination, a potential problem arises if the wide area connectivity of the node running the failure agent is down. Then, the failure agent cannot work properly. We address this issue by running the failure agent on the control node for the whole processing. The control node is responsible for submitting the computation and triggering the wide-area data transfers via third party transfers. If connectivity of control

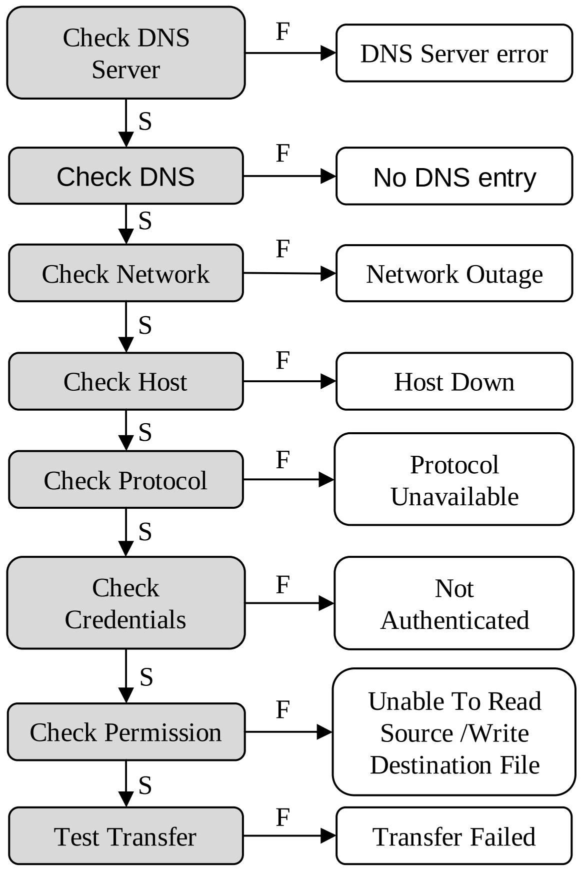


Figure 6.3: Classification of Failures by the Failure Agent

node is down, it cannot start transfers and our scheme works well by marking that as wide-area network outage.

If the host network is reachable, failure agent tries to check if the host is up. If the host is down, it reports that.

After reaching the host, it tests if that protocol is available on that host. If all that works fine, for both hosts, it could be some problem with credential or some misconfiguration. The failure agent tries to authenticate that user to the system and sees if that goes fine. If it fails, it is an authentication problem.

After authentication, it checks for the access to source file and ability to write to destination file. If any fail, it logs the appropriate error. If the system gives enough feedback, the failure agent tries to differentiate between source file not existing and lack of permission to access the source file. Similarly, for destination file, it tries to distinguish between being unable to create the destination file, lack of permission to write to destination file and being unable to write any data to the destination file. The next step may be to try to transfer some dummy data to see if the server works. Optionally, this part can use a test suite that can test a data transfer service.

Authenticating the user, checking permission and running test transfers requires that the failure agent has the user credentials and the failure agent handles this by interacting with the data placement scheduler.

The user can specify policies that influence the working of the failure agent. For instance, users can specify the order of preference for methods to probe if the host is reachable. At times, users may have an unconventional setup that may confuse the failure agent and the policy mechanism allows sophisticated users to tune it to handle those cases. An example is a packet filter that is set to drop all probe packets.

Failure Manager

The failure manager is responsible for coming up with a strategy to handle transient failures. Users can influence these decisions that failure manager makes by specifying policies. Many storage systems have maintenance windows and the user policy can specify that. For instance, if the user specifies that the storage server has a maintenance window every Sunday between 4 a.m. and 8 a.m., then if the storage server is unreachable during that time, the failure manager would retry the transfer after the maintenance window. Further, some users want the system administrator notified via email if a node is down and they can specify that in the policy. Some users may have hard limits for certain jobs i.e. they want the job completed within a time limit and they do not want the system to execute that job after the time limit. The user may specify this as a policy. Users can also tune the exponential back off strategy and can even explicitly state a strategy for the different transient failures. If users do not want to specify the policy, they can tune the provided policy to their preference.

If the failure manager stores information about previous failures, it can use it to adapt its strategy on the fly. For instance, the first strategy chosen by the failure manager may not be good enough if the failure occurs again and using history, it can find out the strategies that worked well and those that did not and use it to refine future strategies. Since maintaining this state in a persistent manner and recovering from crashes considerably increases the complexity of the failure manager, we have enhanced the Grid knowledgebase to store the failure information. An advantage of this is that different failure managers can share the knowledge about failure enabling each to make better decision. Keeping the failure manager stateless simplifies its design and makes crash recovery simple.

For permanent failures, we need to either consult a higher-level planner or pass the failure to the application.

Checking Data Integrity

Even though the data transfer may have completed successfully, the transferred file may have been corrupted. The only way of verifying this is through end-to-end checksum, i.e. compute source checksum(if it is not already available) and destination checksum and verify that they both match. If we cannot run computation on the destination storage server, we may need to download the written data to a nearby node and compute checksum on it and use that as the destination checksum.

As there is a significant cost associated with checksum, some users may not want to perform checksum on all the data. To help them, we have a data integrity manager that allows users to specify preference on the percentage of data they are willing to checksum.

The data integrity manager turns on checksum for certain of the transfers and does this in a statistically unbiased manner.

Whenever a transfer fails a checksum, the data integrity manager figures out the cause of the data corruption and takes a suitable action based on user specified policy. For instance, if a compute node caused the data corruption, a conservative policy may be to recompute all data generated since the previously verified checksum. Another policy may be to try to do a binary search, by recomputing the results at different points and comparing the checksum of the result with that from the corrupted node. This may help us get a smaller window where the node started corrupting the data. It also depends to certain extent on the type of failure. Both the policies may not work if the node corrupts only some of the computation data, with the conservative being better.

The integrity manager can send out email to the appropriate party about the source of data corruption. It can also feed the information to the job policy manager to avoid the repeat of the problem. For instance, if a computation node corrupts data, it will make sure that jobs do not run on that node again until it is fixed.

For users who do not want to perform checksums but want to verify that all of the data has been transferred, we provide an option that verifies that the source and destination file sizes are same in addition to checking that success is returned by the protocol. We did this when we encountered protocol bugs with certain protocol that return success when only a part of the data has been transferred. This occurred in SRB protocol when the destination disk got full. Users can use this as an optimization before checking checksum, as the system does not have to compute the checksum when the source and destination file sizes do not match.

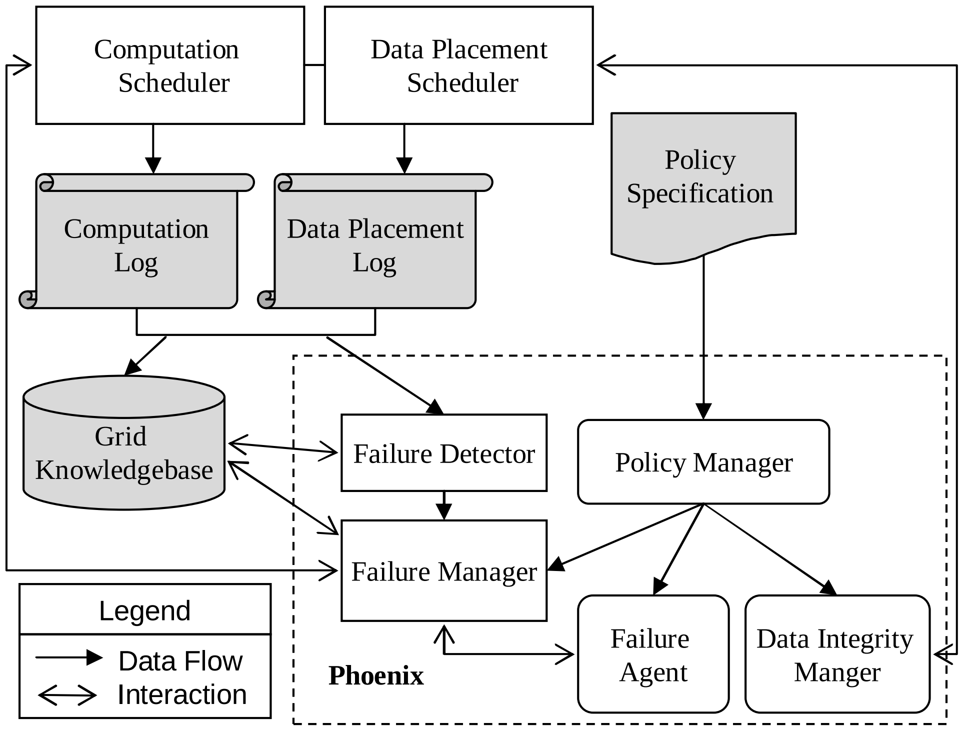


Figure 6.4: Interaction Between Different Phoenix Components

6.6 Putting the Pieces Together

Figure 6.4 shows the overview of how the different pieces of fault-tolerant middleware fit together.

The failure detector scans the user log files of computation scheduler and data placement scheduler to detect failures. It interacts with the Grid Knowledgebase to detect hung transfers and run-away computation. After detecting failures, it passes that information to the failure manager. For data placement failures, the failure manager consults the failure agent to find out the cause of the failure. The failure agent identifies the cause and classifies the data placement failures taking into account user specified policies acquired from the policy manager. The failure manager consults the policy manager and comes up with a strategy to handle the transient failures. It also logs the failure status information to Grid knowledgebase to share that information with other failure managers and to build history to adapt itself.

The data integrity manager based on user policy turns on file size verification and checksum computation and verification for a certain percentage of the data transfers. When a transfer fails the file size verification or checksum verification, it interacts with the data placement scheduler to re-transfer that data.

The failure detector, failure manager, failure agent, policy manager and data integrity together constitute Phoenix, our fault-tolerant middleware layer.

Logging the failures and the strategy taken lets users know the failures encountered. This is useful to address the earlier mentioned information loss when lower layers handle the faults encountered. So, a sophisticated higher layer, either the application or a smart layer between Phoenix and application can use this information to tweak policies of Phoenix and may even convert Phoenix policies into mechanisms by applying the infokernel [23] approach.

While the components we designed and implemented can be easily integrating into existing systems, we found that many users wanted a full-system solution. To address this we designed and implemented a framework that integrates our fault tolerance components.

Figure 6.5 shows the interaction of Phoenix with the components in the system. The user submits a DAG specifying the different jobs and the dependencies between jobs to DAGMan and specifies the policy in ClassAd format [2, 38, 92]. DAGMan submits the computation jobs to Condor/Condor-G and data placement jobs to Stork.

Phoenix keeps monitoring Condor and Stork user log files to detect failures. It uses the Grid Knowledgebase to extract the history and uses it to detect hung transfers and computation.

Taking into account user specified policies, Phoenix classifies failures into transient and permanent and comes up with a suitable strategy to handle transient failures.

It logs both the failure and the strategy taken to the Grid Knowledgebase. Logging the failure allows users to query the Grid Knowledgebase for encountered failures. Phoenix can

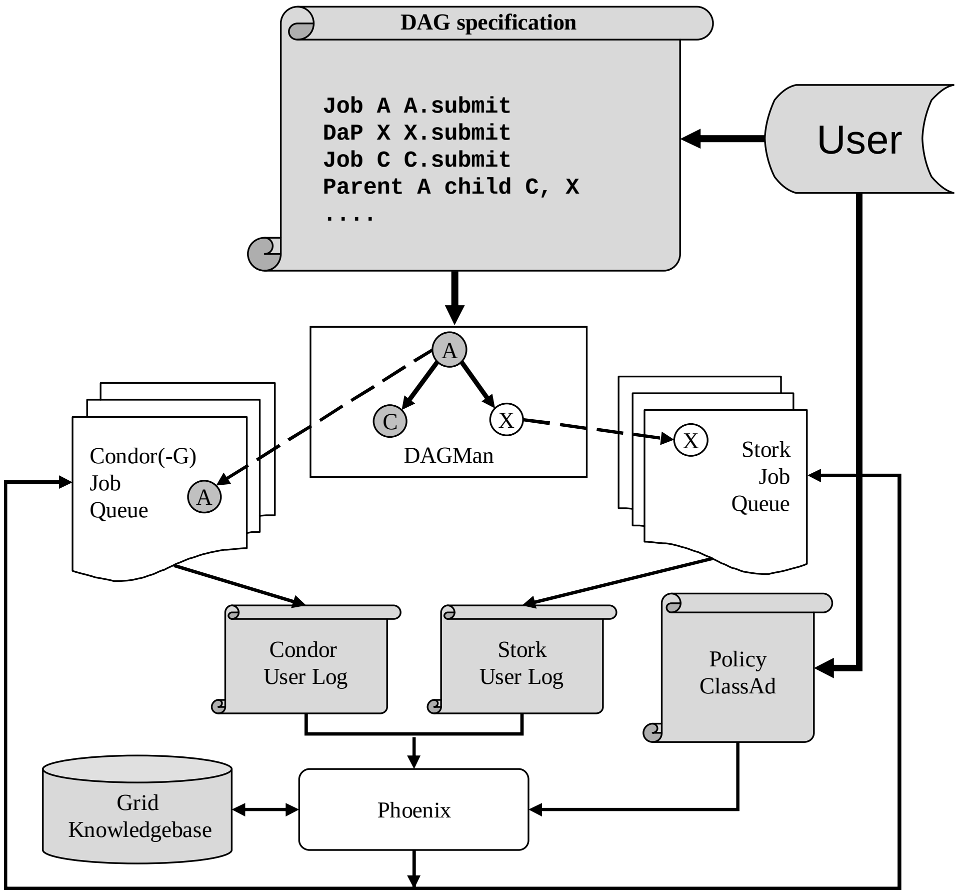


Figure 6.5: Phoenix in the Big Picture

potentially use this to adapt its strategy on the fly.

Phoenix can also turn on checksum and file size verification in a statistically unbiased manner for the specified percentage of transfers. In the current form, it does not support checksum if the destination file server does not allow checksum computation to be performed. The difficulty is to come up with a suitable host to transfer the data and verify the checksum and to detect, in a low overhead manner, if this host is corrupting the data. At present, Phoenix passes permanent failures to the application.

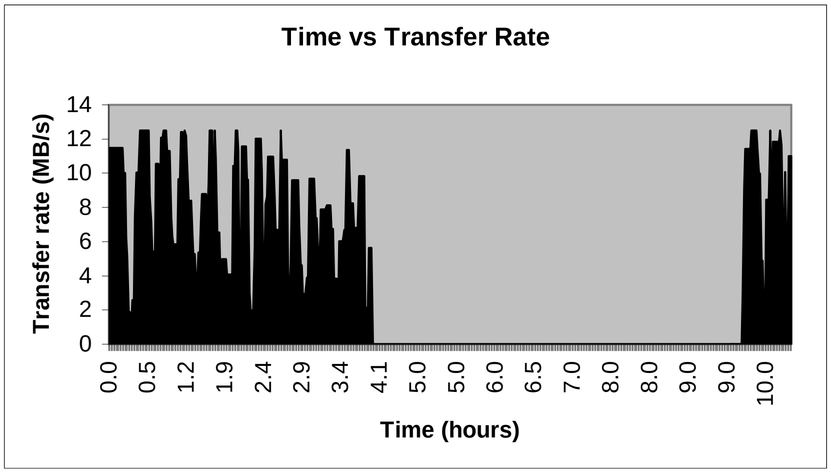


Figure 6.6: Recovery from a Data Transfer Failure

6.7 Insights from NCSA Image Processing Pipeline

The NCSA image processing pipeline prototype involved moving 2611 1.1 GB files(around 3 terabytes) data from SRB mass storage system at San Diego Super Computing Center, CA to NCSA mass storage system at Urbana-Champagne, IL and then processing the images using the compute resources at NCSA, Starlight Chicago and UW-Madison.

During the processing, there was an SRB maintenance window of close to 6 hours. The figure 6.6 shows the pipeline recovering from this transient failure.

Figure 6.7 gives information about 354 data transfers each transferring a different 1.1 GB file with 10 transfers proceeding concurrently. It shows the cumulative distribution of the data transfer times and the number of jobs executing concurrently and the number of jobs completed over a 120-hour period.

Ignoring the outliers, most of the transfers take less than 30 minutes with a standard deviation of 9 minutes. Of the 6 outliers, 3 outliers take 2 1/2 hours each and other three vary between 12 to 96 hours. A simple kill and restart of the outlier data transfer would have resulted in those transfer completing much earlier. Such variations happen because of Heisenbugs [55] and using Phoenix, we can detect them early and retry them to success considerably improving the overall throughput.

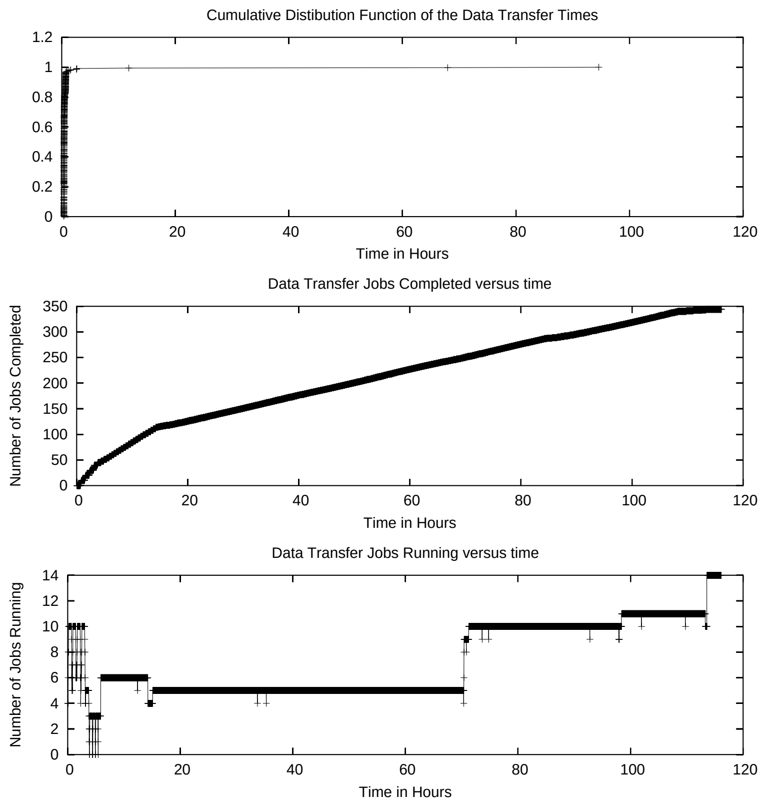


Figure 6.7: Detection of Hung Data Transfers (Heisenbugs)

6.8 Discussion

We have designed and implemented a fault tolerant middleware layer, Phoenix, that transparently makes data intensive distributed applications fault-tolerant. Its unique feature includes detecting hung transfers and misbehaving machines, classifying failures into permanent, transient, and coming up with suitable strategy taking into account user specified policy to handle transient failures. This middleware also handles information loss problem associated with building error handling in lower layers by persistently logging failures to the Grid Knowledgebase and allowing sophisticated application to use this information to tune it.

Chapter 7

Related Work

There has been several work on data placement, providing unified interface to different storage systems, profiling data transfers, run-time adaptation and data pipelines. But to the best of our knowledge, our work is the first to propose that data placement should be regarded as a first class citizen in distributed computing systems, and it requires a specialized system for reliable and efficient management and scheduling.

7.1 Data Placement

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 [46].

The Reliable File Transfer Service (RFT) [83] 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 [20], which is a secure and reliable data transfer protocol especially developed for high-bandwidth wide-area networks.

The Lightweight Data Replicator (LDR) [68] can replicate data sets to the member sites of a Virtual Organization or DataGrid. It was primarily developed for replicating LIGO [9] data, and it makes use of Globus [49] 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) [97] on top of them. Currently, a couple of data storage systems, such as HPSS [7], Jasmin [32] and Enstore [5], support SRMs on top of them. SRMs can also manage distributed caches using "pinning of files".

The SDSC Storage Resource Broker (SRB) [27] 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.

Beck et. al. introduce Logistical Networking [28] 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.

GFarm [87] 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 [76] 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.

7.2 Profiling Data Transfers

CPU, memory and I/O characteristics of commercial and scientific workloads have been well studied [26] [79] [39] [77]. However, storage servers and data transfer protocols in heterogeneous distributed systems have not been profiled and characterized in detail.

Networked Application Logger (NetLogger) [57] toolkit enables distributed applications to precisely log critical events and thereby helps to identify system bottlenecks. It requires application instrumentation, which is difficult for complex and binary-only applications. It cannot be used to log frequent short events and kernel operations. The instrumentation may change the behavior of the program. Since, we wanted to perform a full system characterization that shows the time spent in kernel, we could not use NetLogger.

Vazhkudai et. al. [107] instrumented GridFTP to log performance information for every file transfer and used it to predict the behavior of future transfers. They found that disk I/O takes up to 30% of the total transfer time and using disk I/O data improves end-to-end grid data transfer time prediction accuracy by up to 4% [108].

Our profiling gives a more complete picture of system performance and we believe that this information can be used to make more accurate predictions.

Silberstein et. al. [98] analyzed the effect of file sizes on the performance of local and wide-area GridFTP transfers and found that files sizes should be at least 10 MB for slow wide-area connections and 20 MB for fast local-area connection in order to achieve 90% of optimal performance, and small files do not benefit from multiple streams because of increased overhead of managing the streams. Our profiling work would help people find values for other parameters to achieve close to optimal performance.

7.3 Runtime Adaptation

Network Weather Service (NWS) [110] 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 [107] found that the network throughput predicted by NWS was much less than the actual throughput achieved by GridFTP. He attributed the reason for it being that NWS by default was using 64KB data transfer probes with normal TCP window size to measure throughput. We wanted our network monitoring infrastructure to be as accurate as possible and wanted to use it to tune protocols like GridFTP.

Semke [96] introduces automatic TCP buffer tuning. Here the receiver is expected to advertise large enough windows. Fisk [47] points out the problems associated with [96] and introduces dynamic right sizing which changes the receiver window advertisement according to estimated sender congestion window. 16-bit TCP window size field and 14-bit window scale option which needs to be specified during connection setup, introduce more complications. While a higher value of the window-scale option allows a larger window, it increases the granularity of window increments and decrements. While large data transfers benefit from large window size, web and other traffic are adversely affected by the larger granularity of window-size changes.

Linux 2.4 kernel used in our machines implements dynamic right-sizing, but the receiver window size needs to be set explicitly if a window size large than 64 KB is to be used. Autobuf [1] attempts to tune TCP window size automatically by performing bandwidth estimation before the transfer. Unfortunately there is no negotiation of TCP window size between server and client which is needed for optimal performance. Also performing a bandwidth estimation before every transfer introduces too much of an overhead.

Fearman et. al [45] introduce the Adaptive Regression Modeling (ARM) technique to forecast data transfer times for network-bound distributed data-intensive applications. Ogura et.

al [89] 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 [34], 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) [30] 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.

Kangaroo [100] tries to achieve high throughput by making opportunistic use of disk and network resources.

7.4 Data Pipelines

In [101], Thain et. al. make a detailed study on six batch-pipelined scientific workloads, and analyze data sharing within the different levels of batch pipelines. BAD-FS [29] 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 prefers to access source data from local disk rather than over a network filesystem. BAD-FS at present considers wide-area data movement as a second class activity, and does not perform any scheduling or optimization towards it. BAD-FS can interact with our system for more reliable and efficient wide area transfers.

Pasquale et al [90] present a framework for operating systems level I/O pipelines for efficiently transferring very large volumes of data between multiple processes and I/O devices.

7.5 Fault Tolerance

Thain et. al. propose the Ethernet approach [104] 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.

Medeiros et al. [85] did a survey of failures in the grid environment and found that 76% of the grid deployers had run into configuration problems and 48% had encountered middleware failures. 71% reported that the major difficulty was diagnosing the failure. Our work looks at only data intensive grid applications and our framework would help in diagnosis of failures.

Hwang and Kesselman [60] propose a flexible framework for fault tolerance in the grid consisting of a generic failure detection service and a flexible failure handling framework. The failure detection service uses three notification generators: heart beat monitor, generic grid server and the task itself, while the failure handling framework handles failures at task level using retries, replication and checkpointing and at workflow level using alternative task, workflow-level redundancy and user-defined exception handling. Their work looks at only computation and does not deal with data placement. While heart-beats may help deal with machines that accept jobs and do nothing, they would have difficultly dealing with compute nodes in a private network and compute nodes behind a firewall.

Gray classified computer bugs into Bohrbugs and Heisenbugs [55]. Bohrbug is a permanent bug whereas Heisenbug is a transient bug that may go away on retry. Our classification of failures into permanent and transient is similar to Gray's classification of bugs.

Arpaci-Dusseau [24] et al. introduce the concept of 'fail-stutter' fault tolerance where a component may operate at reduced performance level in addition to failing and stopping. They proposed this model to model components that do not fit under fail-stop model but do not require the generality of Byzantine model.

7.6 Workflow Management

GridDB [81] is a grid middleware based on a data-centric model for representing workflows and their data. GridDB provides users with a relational interface through a three-tiered programming model combining procedural programs (tier 1) and their data through a functional composition language (tier 2). The relational interface (tier 3) provides an SQL-like query and data manipulation language and data definition capability. GridDB allows prioritization of parts of a computation through a tabular interface. GridDB is at a higher-level than our system. It presents a data-centric view to the user and uses the Condor [80]/Condor-G [52] batch scheduling system underneath. Since our system is a transparent layer above Condor/Condor-G, GridDB can easily use our system for data intensive applications and benefit from improved throughput, fault-tolerance, and failure handling.

Chimera [51] is a virtual data system for representing, querying, and automating data derivation. It provides a catalog that can be used by application environments to describe a set of application programs ("transformations"), and then track all the data files produced by executing those applications ("derivations"). Chimera contains a mechanism to locate the "recipe" to produce a given logical file, in the form of an abstract program execution graph. The Pegasus planner [40] maps Chimera's abstract workflow into a concrete workflow DAG that the DAGMan [4] meta-scheduler executes. DAGMan is a popular workflow scheduler and our system addresses the deficiencies of DAGMan and provides new capabilities that considerably improve fault-tolerance, increase throughput, and greatly enhance user experience. Thus, Pegasus and Chimera would benefit from our system.

Chapter 8

Conclusion

The never-ending increase in the computation and data requirements of scientific applications has necessitated the use of widely distributed compute and storage resources to meet the demand. In such an environment, data is no more locally accessible and has thus to be remotely retrieved and stored. Efficient and reliable access to data sources and archiving destinations in a widely distributed environment brings new challenges. Placing data on temporary local storage devices offers many advantages, but such "data placements" also require careful management of storage resources and data movement, i.e. allocating storage space, staging-in of input data, staging-out of generated data, and de-allocation of local storage after the data is safely stored at the destination.

Existing systems closely couple data placement and computation, and consider data placement as a side effect of computation. Data placement is either embedded in the computation and causes the computation to delay, or performed as simple scripts which do not have the privileges of a job. In this dissertation, we propose a framework that de-couples computation and data placement, allows asynchronous execution of each, and treats data placement as a full-fledged job that can be queued, scheduled, monitored and check-pointed like computational jobs. We regard data placement as an important part of the end-to-end process, and express this in a workflow language.

As data placement jobs have different semantics and different characteristics than computational jobs, not all traditional techniques applied to computational jobs apply to data placement jobs. We analyze different scheduling strategies for data placement, and introduce a batch scheduler specialized for data placement. This scheduler implements techniques specific to queuing, scheduling, and optimization of data placement jobs, and provides a level of abstraction between the user applications and the underlying data transfer and storage resources.

We provide a complete data placement subsystem for distributed computing systems, similar to I/O subsystem in operating systems. This system offers transparent failure handling, reliable, efficient scheduling of data resources, load balancing on the storage servers, and traffic control on network links. It provides policy support, improves fault-tolerance and enables higher-level optimizations including maximizing the application throughput. Through deployment in several real-life applications such as US-CMS, DPOSS Astronomy Pipeline', and WCER Educational Video Pipeline, our approach has proved to be effective, providing a promising new research direction.

In this chapter, we will first provide a short summary of the dissertation, and then discuss possible further research directions.

8.1 Summary

Profiling Data Transfers

We have performed a detailed profiling study of data transfer protocols and storage servers. We have examined the effect of concurrency level, and some protocol parameters such as number of parallel streams, and I/O block size on server and client CPU load and data transfer rate. We have made clear the trade-off between single client performance and server load and shown how client performance can be increased and server load decreased at the same time and explained the reason behind this. This allows users to configure and optimize their systems for better end-to-end transfer performance and higher throughput.

The results of this profiling study can be used by the batch schedulers in order to make better scheduling decisions specific to the needs of particular data transfer protocols or data storage servers. This allows us to increase the throughput of the whole system, and decrease the response time for the individual applications. Controlling the CPU server load can also increase reliability of the whole system by preventing server crashes due to overloads.

Scheduling Data Placement

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 designed, implemented and evaluated the first batch scheduler specialized in data placement: Stork. Stork provides a level of abstraction between the user applications and the underlying data transfer protocols; allows queuing, scheduling, and optimization of data placement jobs. We have introduce several possible scheduling strategies for data placement, discuss their similarities and differences compared to the scheduling strategies for computational jobs, and evaluate them.

We have designed, implemented, and evaluated a complete *data placement subsystem* for distributed computing systems, similar to the I/O subsystem in operating systems. This subsystem includes the specialized scheduler for data placement (Stork), a higher level planner aware of data placement jobs, a knowledgebase which can extract useful information from history logs, a failure detection and classification mechanism (Phoenix), and some runtime 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. With several case studies, we have shown the applicability and contributions of our data placement subsystem.

Run-time Adaptation of Data Placement

We have shown a method to dynamically adapt data placement jobs to the environment at the execution time. We have designed and developed a set of disk and memory and network profiling, monitoring and tuning tools which can provide optimal values for I/O block size, TCP buffer size, and the number of TCP streams for data transfers. These values are generated dynamically and provided to the higher level data placement scheduler, which can use them in adapting the data transfers at run-time to existing environmental conditions. We also have provided dynamic protocol selection and alternate protocol fall-back capabilities to provide superior performance and fault tolerance.

Learning from History

We have introduced the concept of Grid knowledgebase that keeps track of the job performance and failure characteristics on different resources as observed by the client side. We have presented the design and implementation of our prototype and evaluated its effectiveness. Grid knowledgebase helped us classify and characterize jobs by collecting job execution time statistics. It also enabled us to easily detect and avoid black holes. Grid knowledgebase has a much wider application area and we believe it will be very useful to the whole distributed systems community. It helps users identify misconfigured or faulty machines and aids in tracking buggy applications.

Failure Detection and Classification

We have designed and implemented a fault tolerant middleware layer that transparently makes data intensive applications in an opportunistic environment fault-tolerant. Its unique feature includes detecting hung transfers and misbehaving machines, classifying failures into permanent, transient, and coming up with suitable strategy taking into account user specified policy to handle transient failures. This middleware also handles information loss problem associated with building error handling in lower layers and allowing sophisticated applications to use this information to tune it.

8.2 Directions for Further Research

In this work, we believe that we have opened a new era in the handling of data placement in widely distributed systems. We cannot claim we have solved all of problems in this area, but we can claim that the conceptual change and the framework we have introduced will change the way researchers approach this problem in the future.

Scheduling

We have introduced a batch scheduler specialized in data placement. Although this scheduler has been tested in several real life applications and has proved to be a reliable and efficient solution, it is open to may improvements. One of the improvements would be in the scheduling strategies. We have applied and evaluated several scheduling strategies in our work, but there remains much to do in this area. Better scheduling strategies can be developed in utilizing the available storage, in controlling the load on the servers or in the network links, and preventing redundant transfers. In this dissertation, we did not even touch on the issues such as selecting the best replica to transfer the data from, caching, and aggregating some of the transfers for better performance. These are all open research issues.

Co-allocation of Resources

Our work has involved interaction of the data scheduler with higher-level planners and CPU schedulers. This interaction is currently not at the desired level it should be. Careful decisions should be made when to allocate storage, when to stage the data, and when to assign the CPU to that job on the execution site in order to achieve efficiency. To achieve this, higher-level planners, the computation and data schedulers should work together and in harmony. Another future research direction is to study the possible ways to enhance this interaction for co-allocation of CPU, storage, and even network resources.

Learning from History

Using the history information in making future scheduling decisions has been a technique used for a long time and we have applied this to our work as well. In our work, we have used the information collected from the clients only and created a client-centric view of the used resources. This technique was effective in providing very useful information for a very low cost. But, it does not provide a universal view of all of the resources available. This could be done by aggregating the information from all of the clients, or using information also from other resources such as execution sites, servers, and network routers. This would increase the information to be interpreted enormously and would require enhanced database management and data mining techniques.

Profiling

In our profiling work, we were be able to study a limited number of data transfer and storage servers. The results we got has provided us sufficient insightful information on the scheduling requirements of these systems. But this study does not cover a broad range of data transfer and storage servers, and not all of the parameters that could be studied. This work can be extended to cover a broader range of systems, and more in-depth analysis in order to find better correlations between different protocol parameters and their impact on performance, reliability and resource utilization. This information can be very useful in designing and developing next generation schedulers which take data placement into consideration when making scheduling decisions.

Bibliography

[1] Auto tuning enabled FTP client and server. http://dast.nlanr.net/Projects/Autobuf/.

[2] Classified Advertisements (ClassAds). http://www.cs.wisc.edu/condor/classad/.

[3] The Compact Muon Solenoid Project (CMS). http://cmsinfo.cern.ch/.

[4] The Directed Acyclic Graph Manager. http://www.cs.wisc.edu/condor/dagman/.

[5] Enstore Mass Storage System. http://www.fnal.gov/docs/products/enstore/.

[6] The Grid2003 production grid. http://www.ivdgl.org/grid2003/.

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

[8] The Large Synoptic Survey Telescope (LSST). http://www.lsst.org/.

[9] Laser Interferometer Gravitational Wave Observatory. http://www.ligo.caltech.edu/.

[10] The National Middleware Initiative (NMI). http://www.nsf-middleware.org/.

[11] NCBI: Growth of genbank. http://www.ncbi.nlm.nih.gov/Genbank/genbankstats.html.

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

[13] A system wide profiler for Linux. http://oprofile.sourceforge.net/.

[14] The Higgs Boson. http://www.exploratorium.edu/origins/cern/ideas/higgs.html.

[15] The TCP/UDP bandwidth measurement tool. http://dast.nlanr.net/Projects/Iperf/.

[16] A Toroidal LHC ApparatuS Project (ATLAS). http://atlas.web.cern.ch/.

[17] The Visible and Infrared Survey Telescope for Astronomy. http://www.vista.ac.uk/.

[18] XML path language (XPath). http://www.w3.org/TR/xpath.html.

[19] Using and administering IBM LoadLeveler. IBM Corporation SC23-3989, 1996.

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

[21] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic Local Alignment Search Tool. *Journal of Molecular Biology*, 3(215):403–410, October 1990.

[22] J. M. Anderson, L. M. Berc, J. Deanand, Sanjay Ghemawat, M. R. Henzinger, S. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurgerand, and W. E. Weihl. Continuous profiling: Where have all the cycles gone? In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, 1997.

[23] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, N. C. Burnett, T. E. Denehy, T. J. Engle, H. S. Gunawi, J.A. Nugent, and F. I. Popovici. Transforming policies into mechanisms with infokernel. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 90–105. ACM Press, 2003.

[24] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. Fail-Stutter fault tolerance. In *The Eighth Workshop on Hot Topics in Operating Systems (HotOS VIII)*, pages 33–38, Schloss Elmau, Germany, May 2001.

[25] A. Avizienis and J.C. Laprie. Dependable computing: From concepts to design diversity. In *Proceeding of the IEEE*, volume 74-5, pages 629–638, May 1986.

[26] L. A. Barroso, K. Gharachorloo, and E. Bugnion. Memory system characterization of commercial workloads. In *Proceedings of the International Symposium on Computer Architecture*, 1998.

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

[28] 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.

[29] 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.

[30] 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.

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

[32] 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.

[33] 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.

[34] 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.

[35] T. L. Casavant and J. G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engineering*, 14(2):141–154, February 1998.

[36] D. Chamberlin. XQuery: An XML query language. *IBM Systems Journal*, 41(4):597–615, 2002.

[37] L. Chervenak, Naveen Palavalli, Shishir Bharathi, Carl Kesselman, and Robert 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.

[38] N. Coleman, R. Raman, M. Livny, and M. Solomon. Distributed policy management and comprehension with classified advertisements. Technical Report UW-CS-TR-1481, University of Wisconsin - Madison Computer Sciences Department, April 2003.

[39] P. E. Crandall, R. A. Aydt, A. A. Chien, and D. A. Reed. Input/output characteristics of scalable parallel applications. In *Proceedings of the IEEE/ACM Conference of Supercomputing*, San Diego, CA, 1995.

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

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

[42] E. W. Dijkstra. The structure of the THE-multiprogramming system. *Communications of the ACM*, 11(5), 1967.

[43] 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.

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

[45] 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.

[46] W. Feng. High performance transport protocols. Los Alamos National Laboratory, 2003.

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

[48] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the Grid: Enabling scalable virtual organizations. *International Journal of Supercomputing Applications*, 2001.

[49] 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.

[50] 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.

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

[52] 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.

[53] E. Gabriel, M. Resch, T. Beisel, and R. Keller. Distributed computing in a heterogeneous computing environment. *Lecture Notes in Computer Science*, 1497:180, January 1998.

[54] S. Ghemawat, H. Gobioff, and S. Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43. ACM Press, 2003.

[55] J. Gray. Why do computers stop and what can be done about them? Technical Report TR-85.7, Tandem, June 1985.

[56] J. Gray and P. Shenoy. Rules of thumb in data engineering. In *Proceedings of the IEEE International Conference on Data Engineering*, San Diego, CA, February 2000.

[57] D. Gunter and B. Tierney. NetLogger: A toolkit for distributed system performance tuning and debugging. In *Integrated Network Management*, 2003.

[58] R. Henderson and D. Tweten. Portable Batch System: External reference specification, 1996.

[59] W. Hoschek, J. Jaen-Martinez, A. Samar, H. Stockinger, and K. Stockinger. Data management in an international DataGrid project. In *First IEEE/ACM Int'l Workshop on Grid Computing*, Bangalore, India, December 2000.

[60] S. Hwang and C. Kesselman. A flexible framework for fault tolerance in the Grid. *Journal of Grid Computing*, 2004.

[61] G. Kola, T. Kosar, and M. Livny. Run-time adaptation of grid data placement jobs. In *Proceedings of Int. Workshop on Adaptive Grid Middleware*, New Orleans, LA, September 2003.

[62] G. Kola, T. Kosar, and M. Livny. Client-centric grid knowledgebase. In *Proceedings of the 2004 IEEE International Conference on Clus ter Computing (Cluster 2004)*, September 2004.

[63] G. Kola, T. Kosar, and M. Livny. A fully automated fault-tolerant system for distributed video processing and off-site replication. In *Proceedings of the 14th ACM International Workshop on Netwo rk and Operating Systems Support for Digital Audio and Video (NOSSDAV 2004)*, Kinsale, Ireland, June 2004.

[64] G. Kola, T. Kosar, and M. Livny. Phoenix: Making data-intensive grid applications fault-tolerant. In *Proceedings of 5th IEEE/ACM International Workshop on Grid Computing (Grid 2004)*, Pittsburgh, PA, November 2004.

[65] G. Kola, T. Kosar, and M. Livny. Profiling grid data transfer protocols and servers. In *Proceedings of 10th European Conference on Parallel Processing (Europar 2004)*, Pisa, Italy, August 2004.

[66] G. Kola, T. Kosar, and M. Livny. Faults in large distributed systems and what we can do about them. In *Proceedings of 11th European Conference on Parallel Processing (Europar 2005)*, Lisbon, Portugal, September 2005.

[67] 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.

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

[69] T. Kosar, G. Kola, and M. Livny. Data-pipelines: Enabling large scale multi-protocol data transfers. In *Proceedings of 2nd International Workshop on Middleware for Grid Computing (MGC 2004)*, Toronto, Canada, October 2004.

[70] T. Kosar, G. Kola, and M. Livny. Reliable, automatic transfer and processing of large scale astronomy datasets. In *Proceedings of 14th Astronomical Data Analysis Software and Systems Conference (ADASS 2004)*, Pasadena, CA, October 2004.

[71] T. Kosar, G. Kola, and M. Livny. Building reliable and efficient data transfer and processing pipelines. *Concurrency and Computation: Practice and Experience*, 2005.

[72] T. Kosar and G. Kola M. Livny. A framework for self-optimizing, fault-tolerant, high performance bulk data transfers in a heterogeneous grid environment. In *Proceedings of 2nd Int. Symposium on Parallel and Distributed Computing (ISPDC 2003)*, Ljubljana, Slovenia, October 2003.

[73] T. Kosar and M. Livny. Stork: Making data placement a first class citizen in the Grid. In *Proceedings of the 24th Int. Conference on Distributed Computing Systems (ICDCS 2004)*, Tokyo, Japan, March 2004.

[74] T. Kosar and M. Livny. A framework for reliable and efficient data placement in distributed computing systems. *Journal of Parallel and Distributed Computing*, 2005.

[75] T. Kosar, S. Son, G. Kola, and M. Livny. Data placement in widely distributed environments. In *Advances in Parallel Computing*, Elsevier Press, 2005.

[76] 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.

[77] S. Kuo, M. Winslett, Y. Cho, J. Lee, and Y.Chen. Efficient input and output for scientific simulations. In *Proceedings of I/O in Parallel and Distributed Systems*, 1999.

[78] L. Lamport and N. Lynch. Distributed computing: Models and methods. *Handbook of Theoretical Computer Science*, pages 1158–1199, Elsevier Science Publishers, 1990.

[79] D. C. Lee, P. J. Crowley, J. L. Bear, T. E. Anderson, and B. N. Bershad. Execution characteristics of desktop applications on Windows NT. In *Proceedings of the 25th International Symposium on Computer Architecture*, 1998.

[80] 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.

[81] D. T. Liu and M. J. Franklin. Griddb: Data-centric services in scientific grids. In *The Second Workshop on Semantics in Peer-to-Peer and Grid Computing*, New York, May 2004.

[82] M. Livny and D. Thain. Caveat emptor: Making grid services dependable from the client side. In *2002 Pacific Rim International Symposium on Dependable Computing (PRDC'02)*, Tsukuba, Japan, December 2002.

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

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

[85] R. Medeiros, W. Cirne, F. Brasileiro, and J. Sauvé. Faults in grids: Why are they so bad and what can be done about it? In *Proceedings of the Fourth International Workshop on Grid Computing*, Phoenix, Arizona, November 2003.

[86] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), 1965.

[87] 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.

[88] 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.

[89] 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.

[90] J. Pasquale and E. Anderson. Container shipping: Operating system support for I/O intensive applications. *IEEE Computer*, 1994.

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

[92] 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.

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

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

[95] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun Network Filesystem. In *Proc. Summer 1985 USENIX Conf.*, pages 119–130, Portland OR (USA), 1985.

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

[97] A. Shoshani, A. Sim, and J. Gu. Storage Resource Managers: Middleware components for Grid storage. In *Nineteenth IEEE Symposium on Mass Storage Systems*, 2002.

[98] M. Silberstein, M. Factor, and D. Lorenz. Dynamo - directory, net archiver and mover. In *Proceedings of the third International Workshop on Grid Computing*, Baltimore, MD, November 2002.

[99] M. Stonebraker and L. A. Rowe. The design of Postgres. In *SIGMOD Conference*, pages 340–355, 1986.

[100] 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.

[101] D. Thain, J. Bent, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Pipeline and batch sharing in grid workloads. In *Proceedings of the Twelfth IEEE Symposium on High Performance Distributed Computing*, June 2003.

[102] D. Thain and M. Livny. Error scope on a computational grid: Theory and practice. In *Proceedings of the Eleventh IEEE Symposium on High Performance Distributed Computing*, Edinburgh, Scotland, July 2002.

[103] D. Thain and M. Livny. Building reliable clients and servers. In Ian Foster and Carl Kesselman, editors, *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2003.

[104] 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.

[105] 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.

[106] C. Thorn. Creating new histories of learning for Math and science instruction: Using NVivo and Transana to manage and study large multimedia datasets. In *Conference on Strategies in Qualitative Research: Methodological issues and practices in using QSR NVivo and NUD*, London, February 2004.

[107] 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.

[108] S. Vazhkudai and J. M. Schopf. Using disk throughput data in predictions of end-to-end grid data transfers. In *Proceedings of the third International Workshop on Grid Computing*, Baltimore, MD, November 2002.

[109] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A note on distributed computing. Technical Report TR-94-29, Sun Microsystems Laboratories, November 1994.

[110] 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.

[111] S. Zhou. LSF: Load sharing in large-scale heterogeneous distributed systems. In *Proc. of Workshop on Cluster Computing*, 1992.