Data Placement in Widely Distributed Environments

T. Kosar[a], S. Son[a], G. Kola[a], and M. Livny[a]

[a]Computer Sciences Department, University of Wisconsin-Madison
1210 West Dayton Street, Madison WI 53706
Email: {kosart, sschang, kola, miron}@cs.wisc.edu

The increasing computation and data requirements of scientific applications, especially in the areas of bioinformatics, astronomy, high energy physics, and earth sciences, have necessitated the use of distributed resources owned by collaborating parties. While existing distributed systems work well for compute-intensive applications that require limited data movement, they fail in unexpected ways when the application accesses, creates, and moves large amounts of data over wide-area networks. Existing systems closely couple data movement and computation, and consider data movement as a side effect of computation. In this chapter, we propose a framework that de-couples data movement from computation, allows queuing and scheduling of data movement apart from computation, and acts as an I/O subsystem for distributed systems. This system provides a uniform interface to heterogeneous storage systems and data transfer protocols; permits policy support and higher-level optimization; and enables reliable, efficient scheduling of compute and data resources.

1. Introduction

The computational and data requirements of scientific applications have increased drastically over the recent years. Just a couple of years ago, the data requirements for an average scientific application were measured in Terabytes, whereas today we use Petabytes to measure them. Moreover, these data requirements continue to increase rapidly every year. A good example for this is the Compact Muon Solenoid (CMS) [1] project, a high energy physics project participating in the Grid Physics Network (GriPhyN). According to the Particle Physics Data Grid (PPDG) deliverables to CMS, the data volume of CMS, which is currently a couple of Terabytes per year, is expected to subsequently increase rapidly, so that the accumulated data volume will reach 1 Exabyte (1 million Terabytes) by around 2015 [2]. This is the data volume required by only one application, and there are many other data intensive applications from other projects with very similar data requirements, ranging from genomics to biomedical, and from metallurgy to cosmology.

The problem is not only the enormous I/O needs of these data intensive applications, but also the number of users who will access the same datasets. For each of the projects, number of people who will be accessing the datasets range from 100s to 1000s. Furthermore, these users are not located at a single site, rather they are distributed all across the country, even the globe. So, there is a prevalent necessity to move large amounts of

data around wide area networks to complete the computation cycle, which brings with it the problem of efficient and reliable data placement. Data 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 compute resources and network resources need to be carefully scheduled and managed, the scheduling of data placement activities all across the Grid is crucial, since the access to data has the potential to become the main bottleneck for data intensive applications. This is especially the case when most of the data is stored on tape storage systems, which slows down access to data even further due to the mechanical nature of these systems.

The common approach to solve this problem of data placement has been either doing it manually, or employing simple scripts, which do not have any automation or fault tolerance capabilities. They cannot adapt to a dynamically changing distributed computing environment. They do not have a single point of control, and generally require babysitting 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 [3].

The Reliable File Transfer Service(RFT) [4] and the Lightweight Data Replicator (LDR) [5] were developed to allow fast and secure replication of data over wide area networks. Both RFT and LDR make use of Globus [6] tools to transfer data, and work only with a single data transport protocol, which is GridFTP [7].

There is an ongoing effort to provide a unified interface to different storage systems by building Storage Resource Managers (SRMs) [8] on top of them. Currently, a couple of data storage systems, such as HPSS [9], Jasmin [10] and Enstore [11], support SRMs on top of them. On the other hand, the SDSC Storage Resource Broker (SRB) [12] 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.

There has also been some efforts to provide reliability and fault tolerance for data placement in distributed systems. Thain et. al. proposed the Ethernet approach [13] 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.

GFarm [14] provided a global parallel filesystem with online petascale storage. OceanStore [15] aimed to build a global persistent data store that can scale to billions of users. BAD-FS [16] was designed as a batch aware distributed filesystem for data intensive workloads.

In this chapter, we present a new approach to handle these problems. This new approach comes with a totally new concept: "Data placement activities must be first class citizens in widely distributed environments just like the computational jobs." They need to be queued, scheduled, monitored, and even check-pointed. It must be made sure that they complete successfully and without any need for human intervention. In other approaches, data placement is generally not considered part of the end-to-end performance, and re-

quires lots of baby-sitting. In this new approach, the end-to-end processing of the data is completely automated, so that the user can just launch a batch of computational/data placement jobs and then forget about it [17].

On the other hand, data placement jobs should be treated differently from computational jobs, since they have different semantics and different characteristics. Data placement jobs and computational jobs should be differentiated from each other and each should be submitted to specialized schedulers that understand their semantics. For example, if the transfer of a large file fails, we may not simply want to restart the job and re-transfer the whole file. Rather, we may prefer transferring only the remaining part of the file. Similarly, if a transfer using one protocol fails, we may want to try other protocols supported by the source and destination hosts to perform the transfer. We may want to dynamically tune up network parameters or decide concurrency level for specific source, destination and protocol triples. A traditional computational job scheduler does not handle these cases. For this purpose, we have developed a "data placement subsystem" for distributed computing systems [18], similar to the I/O subsystem in operating systems. This subsystem includes a specialized scheduler for data placement, a higher level planner aware of data placement jobs, a resource broker/policy enforcer and some optimization tools.

In addition to the scheduling and management of data placement activities, the heterogeneous and multi-administrative nature of the widely distributed systems introduces several other problems. Different storage systems and data transfer protocols need to be accessed, firewalls need to be passed, network fluctuations need to be considered, and all kinds of failures need to be handled. In this chapter, we also discuss how we address all of these problems. We start by highlighting the challenges for data placement activities in widely distributed systems.

2. Data Placement Challenges in Widely Distributed Systems

The widely distributed environments provide researchers with enormous resources, but they also bring some challenges with them. Below are some of the data placement related challenges we will be addressing in this chapter.

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

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

Instead, we should be able to hide these from the application by a mediating system.

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

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

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

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

Changing Conditions. Many tunable parameters depend on the current state of the network, server, and other components involved in the pipeline. Ideally, the system should be able to figure this out and adapt the application. A low-level example is that the TCP buffer size should be set equal to the bandwidth delay product to utilize the full bandwidth. A higher-level example is that to maximize throughput of a storage server, the number of concurrent data transfers should be controlled taking into account server, end host, and network characteristics. Current systems do not perform automated tuning.

Efficient Utilization of Available Bandwidth. Wide-area network bandwidth is increasing. Unfortunately, many of the applications are unable to use the full available bandwidth. New data transport protocols are capable of using almost the entire bandwidth, but tuning them to do so is difficult. Further, users want the ability to give different bandwidth to different applications. Currently, this is very difficult to accomplish.

Connectivity. Firewalls/NATs provide many benefits such as network protection, a solution to IPv4 address shortage, and easy network planning. However, these devices come with prices as well, notably non-universal (and asymmetric) connectivity of the Internet. Because of the connectivity problem, a data may not be moved to a desired

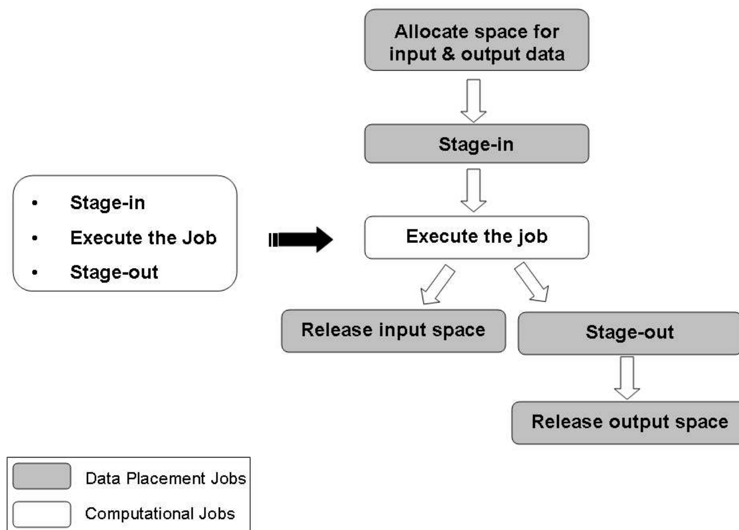

Figure 1. Separating data placement form computation. *Computation at a remote site with input and output data requirements can be achieved with a new five-step plan, in which computation and data placement are separated. This is represented as a six node DAG in the figure.*

place, no matter how a scheduler tries hard. Reconnecting and trying other protocol may not help in this case. For computational jobs, the problem can be mitigated by extra movements of jobs to the places that the final destination can communicate with. However, the extra movements may not be affordable to the data placement, considering the size of data being moved. Even if no extra copies need to be made, data must flow some strategic points to reach a final destination. The constraints imposed by underlying fabric may make data placement efforts we will discuss less effective or even useless.

Several traversal systems that enable applications to communicate over firewalls/NAT have been developed. However, each system has a unique security and other characteristics and supports only its intended use cases. Organizations use different firewalls/NATs and have different network settings, security concerns, performance requirement, etc. Different organizations want to or have to allow traffic into their networks in different ways. In data placement, one site must be able to communicate with multiple sites over time or in a single moment. This multi-organizational and multi-party communication pattern makes previous traversal systems almost useless.

In order to utilize the resources in the widely distributed environments efficiently, researchers have to overcome these challenges first.

3. A New Concept

Most of the data intensive applications in widely distributed environments require moving the input data for the job from a remote site to the execution site, executing the job,

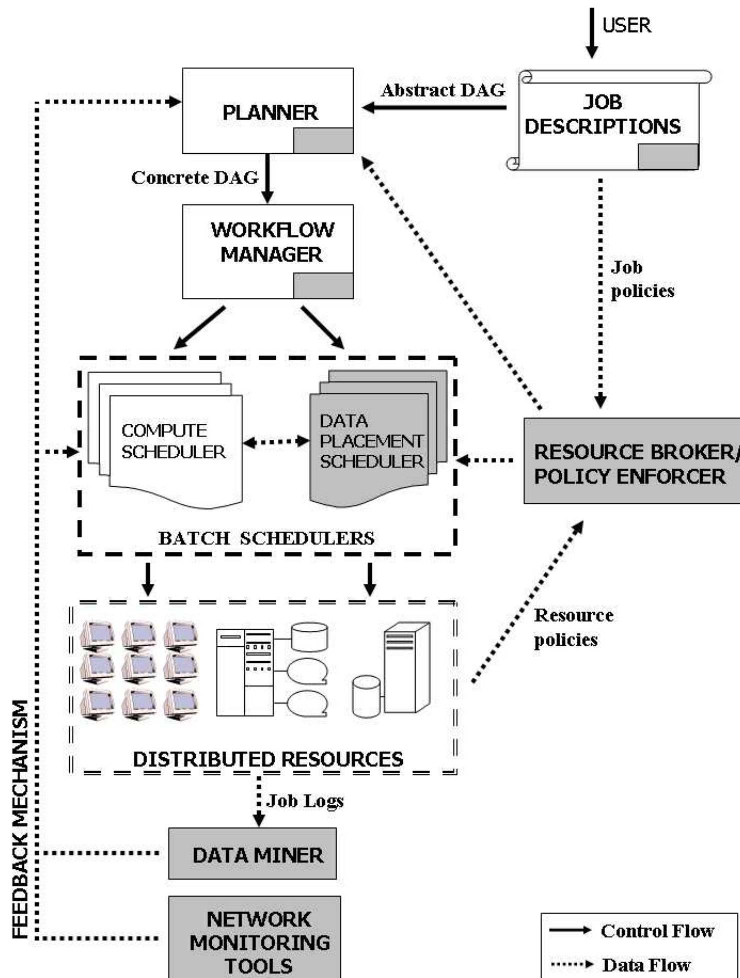

Figure 2. Components of the Data Placement Subsystem. *The components of our data placement subsystem are shown in gray color in the figure.*

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 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 [19], Chimera [20]) 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 [21]). The

workflow manager submits computational jobs to a compute job queue, and the data placement jobs to a data placement job queue. Jobs in each queue are scheduled by the corresponding scheduler. Since our focus in this work is on the data placement part, we do not get into details of the computational job scheduling.

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

The data placement scheduler schedules the jobs in its queue according to the information it gets from the workflow manager and from the resource broker/policy enforcer. The resource broker matches resources to jobs, and helps in locating the data and making decisions such as where to move the data. It consults a replica location service (i.e. RLS [22]) whenever necessary. The policy enforcer helps in applying the resource specific or job specific policies, such as how many concurrent connections are allowed to a specific storage server.

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

The network monitoring tools collect statistics on maximum available end-to-end bandwidth, actual bandwidth utilization, latency and number of hops to be traveled by utilizing tools such as Pathrate [23] and Iperf [24]. Again, the collected statistics are fed back to the scheduler and the resource broker/policy enforcer.

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

4. A New Scheduler: Stork

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

Interaction with Higher Level Planners. Stork can interact with higher level planners and workflow managers. This allows the users to be able to schedule both CPU resources and storage resources together. We made some enhancements to DAGMan, so that it can differentiate between computational jobs and data placement jobs. It can then submit computational jobs to a computational job scheduler, such as Condor [25] or Condor-G [26], and the data placement jobs to Stork. Figure 3 shows a sample DAG specification file with the enhancement of data placement nodes, and how this DAG is

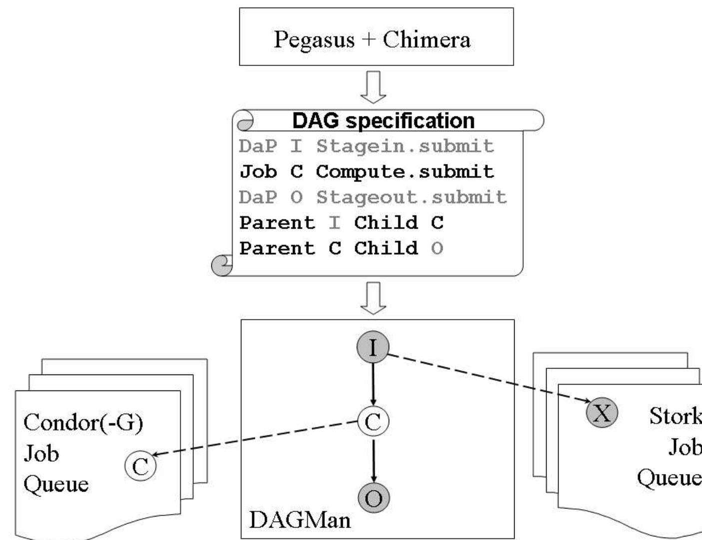

Figure 3. Interaction with Higher Level Planners. *Our data placement scheduler (Stork) can interact with a higher level planners and workflow managers. A concrete DAG created by Chimera and Pegasus is sent to DAGMan. This DAG consists of both computational and data placement jobs. DAGMan submits computational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

handled by DAGMan.

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

Interaction with Heterogeneous Resources. Stork acts like an I/O control system (IOCS) between the user applications and the underlying protocols and data storage servers. It provides complete modularity and extendibility. The users can add support for their favorite storage system, data transport protocol, or middleware very easily. This is a very crucial feature in a system designed to work in a heterogeneous distributed environment. The users or applications may not expect all storage systems to support the same interfaces to talk to each other. And we cannot expect all applications to talk to all the different storage systems, protocols, and middleware. There needs to be a negotiating system between them which can interact with those systems easily and even translate different protocols to each other. Stork has been developed to be capable of this. The modularity of Stork allows users to insert a plug-in to support any storage system, protocol, or middleware easily.

Stork already has support for several different storage systems, data transport protocols, and middleware. Users can use them immediately without any extra work. Stork can interact currently with data transfer protocols such as FTP [27], GridFTP [28], HTTP

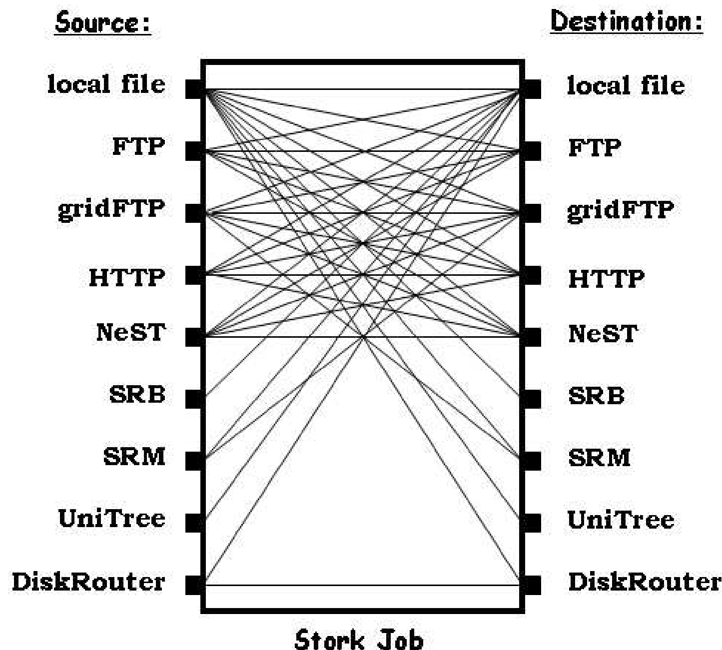

Figure 4. Protocol Translation using Stork Memory Buffer or Third-party Transfers. *Transfers between some storage systems and protocols can be performed directly using one Stork job via memory buffer or third-party transfers.*

and DiskRouter [29]; data storage systems such as SRB [12], UniTree [30], and NeST [31]; and data management middleware such as SRM [8].

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

In order to transfer data between systems for which direct inter-protocol translation is not supported, two consecutive Stork jobs can be used instead. The first Stork job performs transfer from the source storage system to the local disk cache of Stork, and the second Stork job performs the transfer from the local disk cache of Stork to the destination storage system. This is shown in Figure 5.

Flexible Job Representation and Multilevel Policy Support. Stork uses the ClassAd [32] job description language to represent the data placement jobs. The ClassAd language provides a very flexible and extensible data model that can be used to represent arbitrary services and constraints.

Figure 6 shows three sample data placement (DaP) requests. The first request is to allocate 100 MB of disk space for 2 hours on a NeST server. The second request is to transfer a file from an SRB server to the reserved space on the NeST server. The

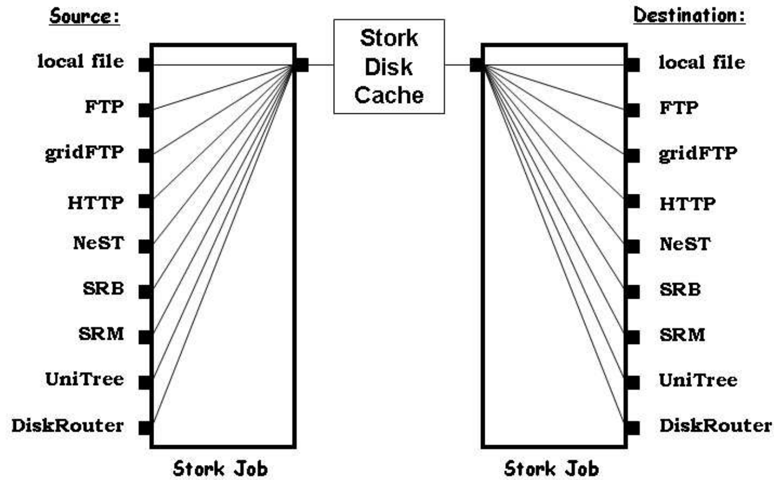

Figure 5. Protocol Translation using Stork Disk Cache. *Transfers between all storage systems and protocols supported can be performed using two Stork jobs via an intermediate disk cache.*

third request is to de-allocate the previously reserved space. In addition to the "reserve", "transfer", and "release", there are also other data placement job types such as "locate" to find where the data is actually located and "stage" to move the data from a tertiary storage to a secondary storage next to it in order to decrease data access time during actual transfers.

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

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

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

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

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

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

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

Figure 6. Job representation in Stork. *Three sample data placement (DaP) requests are shown: first one to allocate space, second one to transfer a file to the reserved space, and third one to de-allocate the reserved space.*

host. Then Stork creates the list of protocols available on each host, and stores these lists as a library:

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

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

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

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

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

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

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

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

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

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

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

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

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

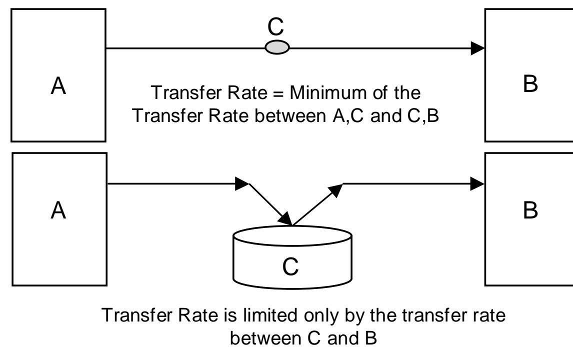

Figure 7. Advantage of buffering at an intermediate node.

5. High Throughput Data Transfers

The steady increase in data sets of scientific applications, the trend towards collaborative research and the emergence of grid computing have created a need to move large quantities of data over wide-area networks. The dynamic nature of network makes it difficult to tune data transfer protocols to use the full bandwidth. Further, data transfers are limited by the bottleneck link and different links become the bottleneck at different times resulting in under-utilization of other network hops. To address these issues, we have designed *DiskRouter*, a flexible infrastructure that uses hierarchical main memory and disk buffering at intermediate points to speed up transfers. The infrastructure supports application-level multicast to reduce network load and enables easy construction of application-level overlay networks to maximize bandwidth.

We present the functions that DiskRouter currently performs.

Store and Forward Device/Wide-area Unix Pipe. DiskRouter in its simplest form is a store and forward device. It uses buffering to match the speed of sender and receiver. It is smart to use main memory first and then disk to perform the buffering. It is slightly different from the normal UNIX pipe in that it provides a tagged block abstraction instead of a continuous stream abstraction. The tagged blocks may arrive out-of-order and the DiskRouter clients at the end-points handle the re-assembly.

Figure 7 shows a case where such a store and forward device improves throughput. A source A is transferring large amounts of data to destination B, and C is an intermediate node between A and C. Placing a DiskRouter at C improves throughput if the bandwidth fluctuation between A and C is independent of the bandwidth fluctuation between C and B. When the bandwidth in the path between A and C is higher than the bandwidth between C and B, data gets buffered at C and when the bandwidth between C and B is higher than the bandwidth between A and C, the buffer drains. Such scenarios occur quite often in real world where A and B are in different time zones and C is an intermediate point.

Data Mover. DiskRouter functions as a data mover. Typically, compute nodes want to get rid of the generated data as quickly as possible and get back to computation. They do not want to spend time waiting for the wide-area transfers to complete and this time

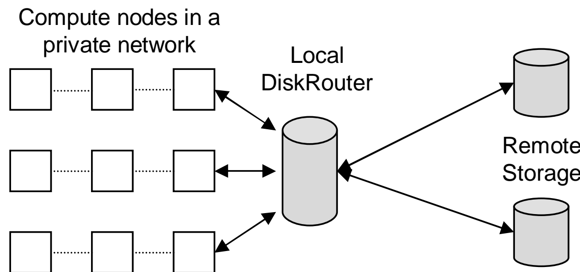

Figure 8. Streaming data via a local DiskRouter.

can be non-deterministic in the presence of failures. In such cases, the computation nodes can write to a local DiskRouter and expect it to take care of pushing the data to the destination. In this function, DiskRouter behaves similar to Kangaroo [33]. It is more efficient, because the data does not have to traverse the disk.

The data mover is very useful when the compute nodes are in a private network and only the head node is accessible outside. In this case, we can deploy DiskRouter on the head-node and use it to stream data to/from the compute nodes. Figure 8 shows this process.

DiskRouter has a significant performance advantage over simply writing the data to disk on the head node and then transferring it because for large amounts of data, disk becomes the bottleneck. Further, the head node may not have enough storage to accommodate all the data. DiskRouter has dynamic flow control whereby it can slow or stop the sender if it runs out of buffer space and make the sender resume sending data when the buffer space becomes available.

Application-level Overlay Network. DiskRouter enables easy construction of application-level overlay network to maximize the throughput of the transfers. While other application-level overlay networks like Resilient Overlay Network (RON) help in reducing latency, DiskRouter overlay-network helps in maximizing bandwidth. Below, we give a concrete example of where this is useful.

In the UW-Madison wide-area network, we have two physical paths to go to Chicago. The direct path has a lower latency but the bandwidth is limited to 100 Mbps. There is an another path to Chicago via Milwaukee which has a bandwidth of 400 Mbps. Unfortunately, because of limitations of current networking(we cannot use two paths and dynamically split data between them), we can use only one path and the current networking based on reducing latency chooses the lower latency (and lower bandwidth) path.

We have been able to deploy a DiskRouter at Milwaukee and exploit the combined bandwidth of 500 Mbps for the data transfer. DiskRouter is able to split the data and dynamically determine the fraction that has to be sent directly and the fraction that has to be sent via Milwaukee. The DiskRouter client reassembles the data and passes the complete data to the application. We find similar cases in other environments as well.

DiskRouter overlay network can also be used to route around failures. Users can build more complex overlay networks and may even dynamically build an overlay network and re-configure it.

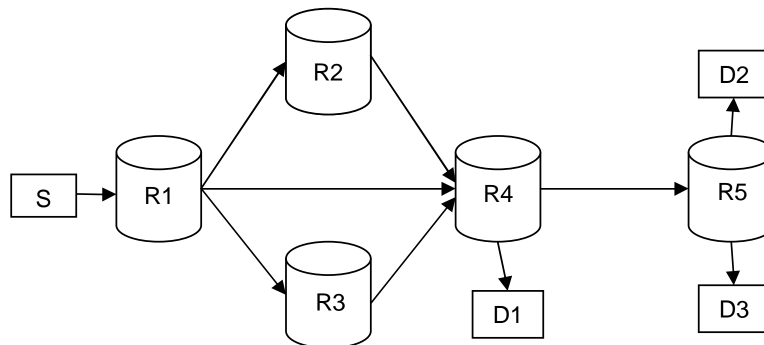

Figure 9. Source S uses a DiskRouter overlay network with DiskRouters R1-R5 to multicast data to destinations D1-D3.

Application-level Multicast. Large collaborative projects have a number of participating sites. The source data needs to be moved to the different participating sites. Some of the participating sites are physically located close by. For example, scientist in NCSA Urbana-Champagne, Illinois and Yale, Connecticut needs the data from Palomar telescope in California.

Unfortunately, IP multicast is not available over this wide-area. The only viable alternative is to build an overlay network with application-level multicast capability. DiskRouter helps accomplish that. Since the DiskRouter has buffering capabilities, not all the endpoints need have the same bandwidth.

In this scheme, a DiskRouter at Chicago would provide the necessary buffering, make copies of the data, and send one copy to NCSA and the other copy to Yale. If terabytes of data are being moved, the network bandwidth saving is quite significant.

Figure 9 shows an application-level multicast overlay network created using DiskRouters R1-R5 for transferring data from source S to destinations D1-D3. The routing here is a little complex. First, the source sends data to DiskRouter R1. R1 dynamically splits the data and sends fractions of the data to R2, R3 and R4. R2 and R3 send all the data received from R1 to R4. R4 sends the data to destination D1 and to the next DiskRouter R5. R5 sends a copy of data to destinations D2 and D3.

Running Computation on Data Streams. DiskRouter allows uploading filters to choose incoming data. Recipients can run choose to run arbitrary computation on the specified amount of data before deciding whether to accept it. Users can use DiskRouters ability to make on-the-fly copies to perform computation on the nodes close to each DiskRouter and then distribute the result through the DiskRouter overlay network. It is also possible to combine data movement and data processing using DiskRouters.

Network Monitor/Dynamic TCP tuning. By using *pathrate* [23] to estimate the network capacity and observing actual transfers, DiskRouter dynamically tunes the TCP buffer size and the number of sockets needed for the transfer to utilize the full bandwidth. Pathrate uses packet dispersion techniques to estimate the network capacity. DiskRouter tunes the buffer size to be equal to the bandwidth delay product. For every 10 ms of latency, DiskRouter adds an extra stream. This is an empirical value that is known to

work well [34]. If multiple streams are being used, the buffer size is split equally among the streams.

It is also possible to regulate the bandwidth used by DiskRouter and the bandwidth used by each DiskRouter stream. At this point, this works by limiting the TCP buffer size.

Since DiskRouter performs this periodic latency and network capacity estimation, it can function as a network monitor.

Integration with Higher Level Planners. DiskRouter has features that enable easy integration with higher-level planners. We believe that this ability is the key to addressing failure and fault tolerance issues. For instance, it is possible when using DiskRouter overlay network that a wide-area network outage disconnects one of the nodes. The DiskRouter client has a tunable time-out and if some pieces are missing after the timeout, it can directly fetch them from the source.

While this handling works, to make better decisions and dynamically reconfigure the overlay network, higher-level planners need this information. DiskRouters and DiskRouter clients pass this information and a summary of the different link status (bandwidth, failures encountered) to higher-level planners, which can then use this information to plan data movement strategies.

In real-world experiments, we have integrated DiskRouter into a data placement framework managed by Stork [17]. Data placement schedulers can make better scheduling decision using the information provided by DiskRouters.

6. Connectivity

Many organizations still resort to a very insecure approach to get around the connectivity problem of the Internet caused by firewalls/NATs. As a data movement becomes necessary, the network administrator manually opens the firewall/NAT for the address quadruples that are believed to be used during the movement. However, it is very difficult to know those quadruples a priori. Especially in the automated data placement, where protocols are dynamically selected and the number of streams is dynamically determined, knowing the exact addresses that a data movement will use is almost impossible. Therefore, to be safe but less secure, the administrator must open more addresses than actually used. These extra openings can be exploited by attackers to sneak into the network.

Previous systems target a specific network setting and/or firewall/NAT behavior. For instance, the approach used by many file sharing software assumes that firewalls/NATs allow web traffic (port 80). Some systems assume that certain components can be installed on firewall/NAT machines or on the boundaries of networks. Organizations use different types of firewalls/NATs and have different network settings, security concerns, performance requirement, etc. Therefore, networks owned by different organizations must be traversed in different ways. For this reason, no previous system satisfies every organization.

In a data placement, one site may have to communicate with multiple sites over time or in a single moment. Those peer sites may have to be traversed in different ways as explained above. Because of this multi-organizational and multi-party communication pattern, we need to have various sorts of traversal systems. Furthermore, we need an

integrated system that combines those systems so that two organizations having different network settings and security criteria can communicate each other. At our best knowledge, UCB (Unified Connection Brokering) is the only system that satisfies these requirements.

UCB is an integrated system that combines 3 component systems, each of which supports unique network settings and use cases. With UCB, an organization can choose one from those component mechanisms and can still talk with others using different component systems. This is very similar to that, in Stork, data can be moved from an organization using a protocol to another using a different protocol. To support as many organizations as possible, we have chosen UCB's component systems very carefully. Instead of adding component systems in ad hoc ways, we classified representative ways to open firewalls/NATs and then invented component systems for each class. The component systems are GCB (Generic Connection Brokering) [35], CODO (Cooperative On-Demand Opening) [36], and XRAY (middleboX traversal by RelAYing).

6.1. Overview

In UCB, each network chooses a component mechanism based upon its network situation and other criteria. Yet, two networks using different component mechanisms can communicate each other. Figure 10 shows a typical topology of UCB. The figure shows a situation that a client in an organization using CODO makes a connection to a server in another organization using GCB.

If a server network does not allow inbound connections, then it has to have one or more agents that arrange connections toward authorized server applications. Similarly, a client network that does not allow outbound connections from arbitrary clients must have one or multiple agents that arrange connections from authorized clients. A UCB agent configured to use GCB as its component mechanism is called a GCB agent. Other types of agents are called similarly. Not every place can an organization install its agents. The places on which agents can (or must) be installed mainly depend on the factors that an organization chooses its component mechanism based upon. For example, GCB must be used when end users (researchers) do not have a control relationship to the firewalls/NATs on a communication path. Since they cannot control the devices, agents cannot be installed on the devices. In this case, researchers normally do not have privilege to install entities on the boundary of the network, either. For this reason, GCB agents are generally installed on the public network as shown in the figure 10.

When a server application creates a socket, UCB library linked with the server (the server library for short) contacts the agent of the server network (the server agent for short) and registers the socket information to the agent. Through this registration process, the agent checks if the server is authorized to accept inbound communications.

To make a connection to a server, a client behind a firewall/NAT contacts the agent of the client network (the client agent for short) and asks for the arrangement of a connection to the server. Next, the client agent, on behalf of the client, contacts the server agent and asks for an inbound connection to the server. If the client network allows outbound connections by arbitrary clients, i.e. the client does not have an agent to arrange outbound connections (as the client in the bottom of figure 10), then the client itself contacts the server agent.

Upon receiving a connection request to a server, the server agent checks if the server

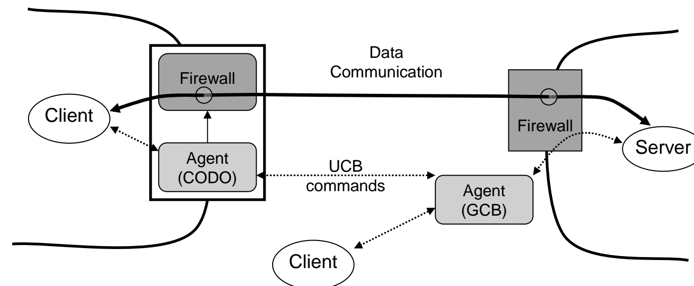

Figure 10. Typical topology of UCB. *A server network on the right has a GCB agent outside of it. A client network on the left has a CODO agent running on the firewall machine.*

and the client (or its agent) are authorized to accept and make inbound connections, respectively. If the test passes, it creates a pinhole or a relay point and replies success to the client agent with the information such as how and to what address the client agent can make the connection.

When the client agent receives the reply from the server agent, it creates a pinhole or a relay point. Then, the client agent notifies the client the result of the arrangement. The server agent also notifies the server, if an action must be triggered by the server. To allow only authorized applications to communicate through a firewall/NAT, a strong security mechanism is used to exchange UCB commands.

The important points of the connection setup process in UCB are (1) that each network opens its firewall/NAT using its component mechanism and (2) that two component mechanisms are glued together through the cooperation of agents.

6.2. Component mechanisms

6.2.1. GCB (Generic Connection Brokering)

GCB is a component system of UCB and supports the cases that end users (usually called researchers in grid) can not control (either manually or programmatically) the firewall/NAT of the server network. For example, if a researcher has a grid pool behind a firewall of the campus network, then (s)he may not be able to control the campus firewall.

Today, most networks allow outbound but block inbound communications. Most NATs allow outbound communications (and their replies). Most firewalls are factory-configured to allow outbound communications. GCB exploits this common configuration. From a different perspective, GCB tries to bring the symmetry back to the Internet. With the traditional socket layer, the first packet, e.g. SYN packet, is always sent from a client to a server. Therefore, the direction of a connection is decided by the role–who the client (or the server) is in a communication. The main idea of GCB is to make the Internet symmetric by decoupling the direction from the role. GCB decides the direction of a connection based upon the relative topology of communicating parties, instead of who calls connect. Because of the decoupling, the role is still decided by application

programmers. Thus, in GCB, it is possible that a connection is made from a server that calls accept to a client calling connect.

To make a connection to a server, a client sends a connection request to the agent of the server. Upon receiving the request, the agent decides the direction of the connection and notifies the client and the server, if necessary. If a client is in the public network and a server is behind a firewall/NAT, the connection is made from the server to the client. If both are behind firewalls/NATs, then both parties make outbound connections to a relay point that the agent creates.

6.2.2. CODO (Cooperative On-Demand Opening)

CODO is used when the firewall/NAT of the network can be programmatically controlled. A CODO agent running on the firewall/NAT machine dynamically creates and deletes pinholes at the device upon requests from applications. CODO controls outbound as well as inbound communications. Therefore, networks need not allow outbound communications to use CODO.

When a client behind a firewall/NAT wants to connect to a server also behind another firewall/NAT, it contacts the client agent and asks for outbound connection toward the server. The client agent then contacts the server agent and asks for inbound connection. Using the API that each firewall/NAT provides, the client and server agents create pinholes for outbound and inbound, respectively, for the quadruple (client's IP, client's port, server's IP, server's port).

Other systems that support similar network settings as CODO creates pinholes with wildcard client address. On the other hand, CODO creates pinholes with full addresses of the client and the server. This results in firewalls/NATs opened as narrow and short as possible, because pinholes are made so that only intended pairs can get through and only when there are authorized client-server pairs. CODO involves more interactions between applications and agents than other systems because it opens firewalls/NATs with more address information. To support organizations that want efficient traversal at the expense of security, it provides efficient but less secure mode, called promiscuous mode. In this mode, CODO operates the same way as other systems.

6.2.3. XRAY (middleboX traversal by RelAYing)

XRAY is used when the firewall/NAT of the network can be manually controlled. The firewall/NAT is configured to allow traffic to/from XRAY agents that are usually installed inside the network. The XRAY agents relay communications for authorized applications, but drop packets from unauthorized ones. Therefore, we can regard that the firewall/NAT trusts the agents and delegates them a part of policy enforcement. Like CODO, XRAY controls both inbound and outbound communications.

A connection is established in a very similar way as with the CODO case. The only difference is that XRAY agents create relay points instead of making pinholes at the firewall/NAT. Application data units are transferred from the client to the relay point in the client network to the relay point in the server network and finally to the server, and vice versa. Relay points terminate overlay link at the application layer. Each overlay link is secured by security negotiation between hops terminating the link. Therefore, every application data unit is authenticated and authorized by each hop.

In addition to the fact that XRAY does not require programmatic control of the fire-

wall/NAT, it can provide the most secure traversal. In CODO, pinholes are made only when authorized applications request so. However, communications through the pinholes are not secured by the traversal mechanism. On the contrary, XRAY creates relay points such that only authorized applications can communicate via those relay points.

One drawback of relay-based approach is the performance. Hop-by-hop encryption/decryption slows down communication. At replay points, packets must traverse the protocol stack up and down. However, these are inevitable costs to achieve secure traversal.

6.2.4. Unifying component mechanisms

To integrate component mechanisms, UCB uses the model that a network uses one mechanism to enable communications into and out of it, while an endpoint uses any mechanism chosen by the network that it wants to communicate into or out of. With this model, we just need to show how UCB connects two mechanisms used by each network to provide an end-to-end communication channel. The followings show how two mechanisms are glued in UCB for each combination of mechanisms used by the client and the server network. Each combination is denoted as "from A to B", where A is the mechanism the client network uses and B used by the server network. We do not have the cases that the client network uses GCB because GCB is not used to control outbound communications.

From CODO to GCB: The agent of the client network (client agent for short) creates a pinhole for inbound from the server to the client. The end-to-end connection is made from the server to the client through the hole.

From CODO to XRAY: The agent of the server network (server agent for short) creates a relay point and the client agent creates a pinhole for outbound from the client to the relay point. The end-to-end connection consists of two connections: one from the client to the relay point (through the hole) and the other from the relay point to the server.

From CODO to CODO: The client and server agent create pinholes for outbound and inbound, respectively, from the client to the server. The end-to-end connection is made from the client to the server.

From XRAY to GCB: The client agent creates a relay point with passive sockets for both client and server sides. The end-to-end connection consists of two connections: one from the client to the relay point and the other from the server to the relay point.

From XRAY to CODO: The client agent creates a relay point and the server agent creates a pinhole for inbound from the relay point to the server. The end-to-end connection consists of two connections: one from the client to the relay point and the other from the relay point to the server through the hole.

From XRAY to XRAY: The client and server agent create relay points. The end-to-end connection consists of three connections: one from the client the relay point in the client network, another from the relay point in the client network to the relay point in the server network, and the other from the relay point in the server network to the server.

7. Conclusions

Existing systems closely couple data movement and computation. This results in recomputation if the output data transfer fails, or re-transfer of the data if the computation fails. This is especially undesirable for data intensive applications, where the transfers

have a higher likelihood of failure due to the large amounts of data moved. We have presented a framework that de-couples data movement from computation, and acts as an I/O subsystem for distributed systems. This system provides a uniform interface to heterogeneous storage systems and data transfer protocols; permits policy support and higher-level optimization; and enables reliable, efficient scheduling of compute and data resources.

The current approaches consider data movement as a side effect of computation. There is no scheduling of data movement. This has resulted in thrashing and crashing of storage servers when there are uncontrolled number of requests to them; uneven and inefficient utilization of the network links; and decreased end-to-end performance of data intensive applications. Just as computation and network resources need to be carefully scheduled and managed, the scheduling of data movement across distributed systems is crucial, since the access to data is generally the main bottleneck for data intensive applications. We have presented an approach which regards data movement as a full-fledged job just like computational jobs. In this approach, data movement can be queued, scheduled, monitored, and even check-pointed. We have also introduced the first specialized scheduler for data movement, which we call Stork.

While work has been done on making computation adapt to changing conditions, little work has been done on making the data movement adapt to changing conditions. Many tunable parameters depend on the current state of the network, server, and other components involved in the data movement. Ideally, the system should be able to figure this out and adapt the application. A low-level example is that the TCP buffer size should be set equal to the bandwidth delay product to utilize the full bandwidth. A higher-level example is that to maximize throughput of a storage server, the number of concurrent data transfers should be controlled taking into account server, end host, and network characteristics. I have presented an infrastructure that observes the environment and enables run-time adaptation of data placement jobs.

We have introduced a flexible infrastructure called DiskRouter that uses hierarchical buffering at intermediate points to aid in large-scale data transfers. It supports easy construction of application level overlay network and can perform routing. It performs dynamic TCP tuning to improve throughput. It supports application level multicast to help lower the network load and improve the system throughput, and it has been integrated into a data placement framework managed by Stork.

Because of its multi-organizational, multi-party, and huge amount of communications, data placement may be the field most damaged from the connectivity problem of the Internet. We have introduced an integrated system, UCB, which handles the problem. Just as Stork is able to handle diverse protocols, middleware, and storage systems, UCB handles the diversity of network settings, types of firewalls/NATs, security concerns of organizations. UCB provides three component mechanisms each of which supports unique use cases and has different characteristics such as performance, security, and deployability. An organization can choose a component mechanism to use based upon its requirements and constraints. Still, organizations using different component mechanisms can communicate each other.

REFERENCES

1. CMS, The Compact Muon Solenoid Project, http://cmsinfo.cern.ch/.
2. PPDG, PPDG Deliverables to CMS, http://www.ppdg.net/archives/ppdg/2001/doc00017.doc.
3. W. Feng, High Performance Transport Protocols, Los Alamos National Laboratory (2003).
4. R. Maddurri, B. Allcock, Reliable File Transfer Service, http://www-unix.mcs.anl.gov/ madduri/main.html (2003).
5. S. Koranda, B. Moe, Lightweight Data Replicator, http://www.lsc-group.phys.uwm.edu/lscdatagrid/LDR/ overview.html (2003).
6. I. Foster, C. Kesselmann, Globus: A Toolkit-Based Grid Architecture, in: The Grid: Blueprints for a New Computing Infrastructure, Morgan Kaufmann, 1999, pp. 259–278.
7. B. Allcock, J. Bester, J. Bresnahan, A. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, S. T. ke, Secure, efficient data transport and replica management for high-pe rformance data-intensive computing, in: IEEE Mass Storage Conference, San Diego, CA, 2001.
8. A. Shishani, A. Sim, J. Gu, Storage Resource Managers: Middleware Components for Grid Storage, in: Nineteenth IEEE Symposium on Mass Storage Systems, 2002.
9. SDSC, High Performance Storage System (HPSS), http://www.sdsc.edu/hpss/.
10. I. Bird, B. Hess, A. Kowalski, Building the mass storage system at Jefferson Lab, in: Proceedings of 18th IEEE Symposium on Mass Storage Systems, San Diego, California, 2001.
11. FNAL, Enstore mass storage system, http://www.fnal.gov/docs/products/enstore/.
12. C. Baru, R. Moore, A. Rajasekar, M. Wan, The SDSC Storage Resource Broker, in: Proceedings of CASCON, Toronto, Canada, 1998.
13. D. Thain, , M. Livny, The ethernet approach to grid computing, in: Proceedings of the Twelfth IEEE Symposium on High Performance Distributed Computing, Seattle, Washington, 2003.
14. Y. Morita, H. Sato, Y. Watase, O. Tatebe, S. Sekiguchi, S. Matsuoka, N. Soda, 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, 2003.
15. J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, 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), 2000.
16. J. Bent, D. Thain, A. Arpaci-Dusseau, 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, 2004.
17. T. Kosar, M. Livny, Stork: Making Data Placement a First Class Citizen in the Grid, in: Proceedings of the 24th Int. Conference on Distributed Computing Systems, Tokyo, Japan, 2004.

18. T. Kosar, M. Livny, A framework for reliable and efficient data placement in distributed computing system s, Journal of Parallel and Distributed Computing.
19. E. Deelman, J. Blythe, Y. Gil, C. Kesselman, Pegasus: Planning for execution in grids, in: GriPhyN technical report, 2002.
20. I. Foster, J. Vockler, M. Wilde, Y. Zhao, Chimera: A virtual data system for representing, querying, and automating data derivation, in: 14th International Conference on Scientific and Statistical Database Management (SSDBM 2002), Edinburgh, Scotland, 2002.
21. D. Thain, T. Tannenbaum, 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.
22. L. Chervenak, N. Palavalli, S. Bharathi, C. Kesselman, R. Schwartzkopf, Performance and scalability of a Replica Location Service, in: Proceedings of the International Symposium on High Performance Distributed Computing Conference (HPDC-13), Honolulu, Hawaii, 2004.
23. C. Dovrolis, P. Ramanathan, D. Moore, What do packet dispersion techniques measure?, in: INFOCOMM, 2001.
24. NLANR/DAST, Iperf: The TCP/UDP bandwidth measurement tool, http://dast.nlanr.net/Projects/Iperf/ (2003).
25. M. J. Litzkow, M. Livny, M. W. Mutka, Condor - A Hunter of Idle Workstations, in: Proceedings of the 8th International Conference of Distributed Computing Systems, 1988, pp. 104–111.
26. J. Frey, T. Tannenbaum, I. Foster, 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, 2001.
27. J. Postel, FTP: File Transfer Protocol Specification, RFC-765 (1980).
28. W. Allcock, I. Foster, R. Madduri, Reliable data transport: A critical service for the grid, in: Building Service Based Grids Workshop, Global Grid Forum 11, 2004.
29. G. Kola, M. Livny, Diskrouter: A flexible infrastructure for high performance large scale data transfers, Tech. Rep. CS-TR-2003-1484, University of Wisconsin (2003).
30. M. Butler, R. Pennington, J. A. Terstriep, Mass Storage at NCSA: SGI DMF and HP UniTree, in: Proceedings of 40th Cray User Group Conference, 1998.
31. Condor, NeST: Network Storage, http://www.cs.wisc.edu/condor/nest/ (2003).
32. R. Raman, M. Livny, 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, 1998.
33. D. Thain, J. Basney, S. Son, 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, 2001.
34. G. Kola, T. Kosar, M. Livny, Run-time adaptation of grid data-placement jobs, Parallel and Distributed Computing Practices.
35. S. Son, M. Livny, Recovering internet symmetry in distributed computing, in: Proceedings of the 3rd International Symposium on Cluster Computing and the Grid, Tokyo, Japan, 2003.

36. S. Son, B. Allcock, M. Livny, Codo: Firewall traversal by cooperative on-demand pening, in: Proceedings of the Fourteenth IEEE Symposium on High Performance Distributed Computing, Research Triangle Park, NC, 2005.