RUN-TIME ADAPTATION OF GRID DATA PLACEMENT JOBS

G. KOLA*, T. KOSAR* & M. LIVNY*

Abstract. Grid presents a continuously changing environment. It also introduces a new set of failures. The data grid initiative has made it possible to run data-intensive applications on the grid. Data-intensive grid applications consist of two parts: a data placement part and a computation part. The data placement part is responsible for transferring the input data to the compute node and the result of the computation to the appropriate storage system. While work has been done on making computation adapt to changing conditions, little work has been done on making the data placement adapt to changing conditions. In this work, we have developed an infrastructure which observes the environment and enables run-time adaptation of data placement jobs. We have enabled Stork, a scheduler for data placement jobs in heterogeneous environments like the grid, to use this infrastructure and adapt the data placement job to the environment just before execution. We have also added dynamic protocol selection and alternate protocol fall-back capability to Stork to provide superior performance and fault tolerance.

Key words. Grid, data placement, run-time adaptation, scheduling, data intensive applications, dynamic protocol selection, stork, condor.

1. Introduction. The grid [10] [11] [19] presents a continuously changing environment. The data grid initiative has increased the underlying network capacity and enabled running of data-intensive applications on the grid. Data-intensive applications consist of two parts: a data placement part and a computation part. The data placement part is responsible for transferring the input data to the compute node and the result of the computation to the appropriate storage system. Data placement encompasses all data movement related activities such as transfer, staging, replication, data positioning, space allocation and deallocation. While work has been done on making computation adapt to changing conditions, little work has been done on making the data placement adapt to changing conditions.

Sophisticated protocols developed for grid data transfers like GridFTP [1] 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

*Department of Computer Sciences, University of Wisconsin-Madison, 1210 W. Dayton St. Madison, WI 53706, USA. ({kola, kosart, miron}@cs.wisc.edu).

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 work, we have developed a monitoring infrastructure which 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 a a data placement scheduler. The data placement scheduler uses the tuned parameters while executing the data placement jobs submitted to it, essentially performing run-time adaptation of data placement jobs. We have also added dynamic protocol selection and alternate protocol fall-back capability to our prototype data placement scheduler. 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.

2. Related Work. Network Weather Service (NWS) [25] 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 [24] 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 [20] introduces automatic TCP buffer tuning. Here the receiver is expected to advertise large enough windows. Fisk [9] points out the problems associated with [20] 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 [15] 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 [8] introduce the Adaptive Regression Modeling (ARM) technique to forecast data transfer times for network-bound distributed data-intensive applications. Ogura et. al [17] 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 [5], 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.

Thain et. al. propose the Ethernet approach [21] to Grid Computing, in which they introduce a simple scripting language which can handle failures in a manner similar to exceptions in some languages. The Ethernet approach is not aware of the semantics of the jobs it is running, its duty is retrying any given job for a number of times in a fault tolerant manner. Kangaroo [22] tries to achieve high throughput by making opportunistic use of disk and network resources.

Application Level Schedulers (AppLeS) [4] have been developed to achieve efficient scheduling by taking into account both application-specific and dynamic system information. AppLeS agents use dynamic system information provided by the NWS.

Beck et. al. introduce Logistical Networking [2] 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.

3. 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 3.1 shows the components of the monitoring and tuning infrastructure and the interaction with the data placement scheduler.

3.1. 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 possible.

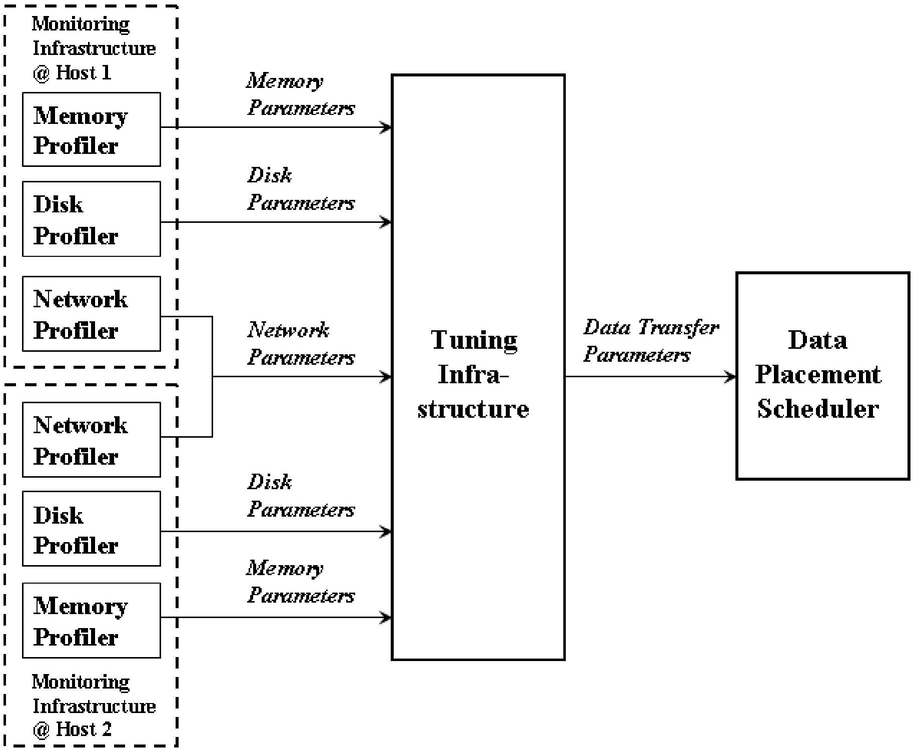

FIG. 3.1. *Monitoring and Tuning Infrastructure. This figure shows an overview of the monitoring and tuning infrastructure. The different profilers determine the various environment conditions and the tuning infrastructure uses that information to generate optimal parameter values.*

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.

3.2. 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.

3.3. Scheduling Data Transfers. The data placement scheduler uses the information provided by the tuning infrastructure to make intelligent decisions for scheduling and executing the data placement jobs.

In our study, we used the Stork [13] data placement scheduler to monitor, manage, and schedule the data transfers over the wide area network. Stork is a specialized scheduler for data placement activities in heterogeneous environments. Stork can queue, schedule, monitor and manage data placement jobs, and it ensures that the jobs complete.

Stork is aware of the semantics of the data placement requests submitted to it, so it can make intelligent scheduling decisions with regard to each individual request. For example, if a transfer of a large file fails, Stork can transfer only parts of the file not already transferred. We have made some enhancements to Stork that enable it to adaptively schedule data transfers at run-time using the information provided by monitoring and tuning infrastructure. These enhancements include dynamic protocol selection and run-time protocol auto-tuning. The details of these enhancements are discussed in section 5.

4. 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.

4.1. 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.

4.2. Network Profiler. The network profiler gets the kernel TCP parameters from /proc. It runs Pathrate [7] 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 [12]. 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 [16] can also be used instead of the DiskRouter data transfer tool to perform the actual transfer. From experience, we found Pathrate to 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. 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 as follows: 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.

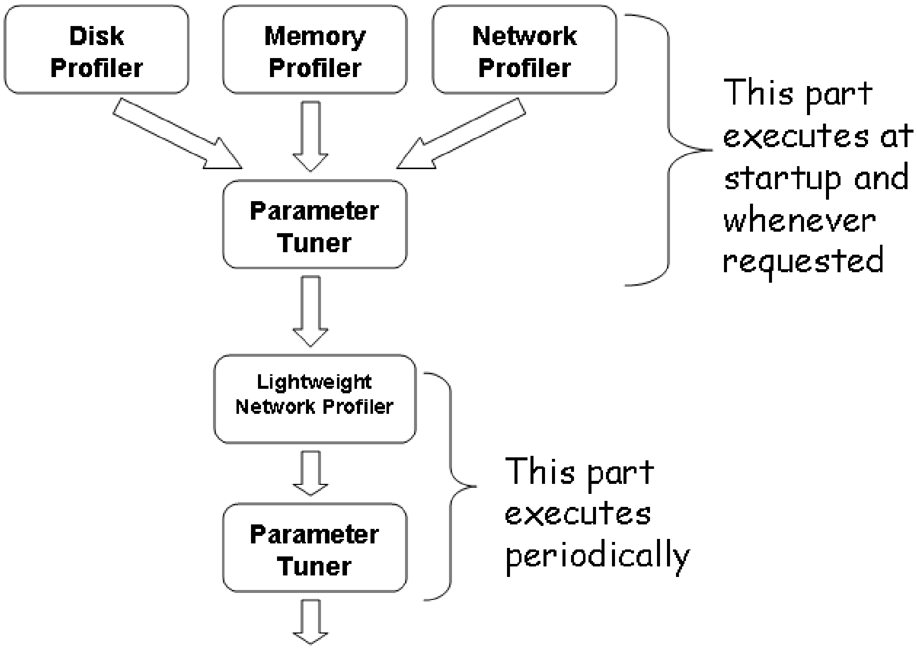

FIG. 4.1. *The DAG Coordinating the Monitoring and Tuning infrastructure. This DAG shows the order in which the monitors(profilers) and tuner are run. Initially all the profilers are run and the information is logged to persistent storage and also passed to the parameter tuner which generates the optimal parameter values. After that, the light-weight network profiler and parameter tuner are run periodically. The parameter tuner uses the values of the earlier profiler runs and the current light-weight network profiler run to generate the optimal parameter values.*

4.4. Coordinating the Monitoring and Tuning Infrastructure. The disk, memory and network profilers need to be run once at startup and the light-weight 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) [6] [23] 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 [14] 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 4.1.

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.

4.5. 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 [18] 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 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 [3] 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.

4.6. 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.

5. 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.

5.1. 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 5.1.

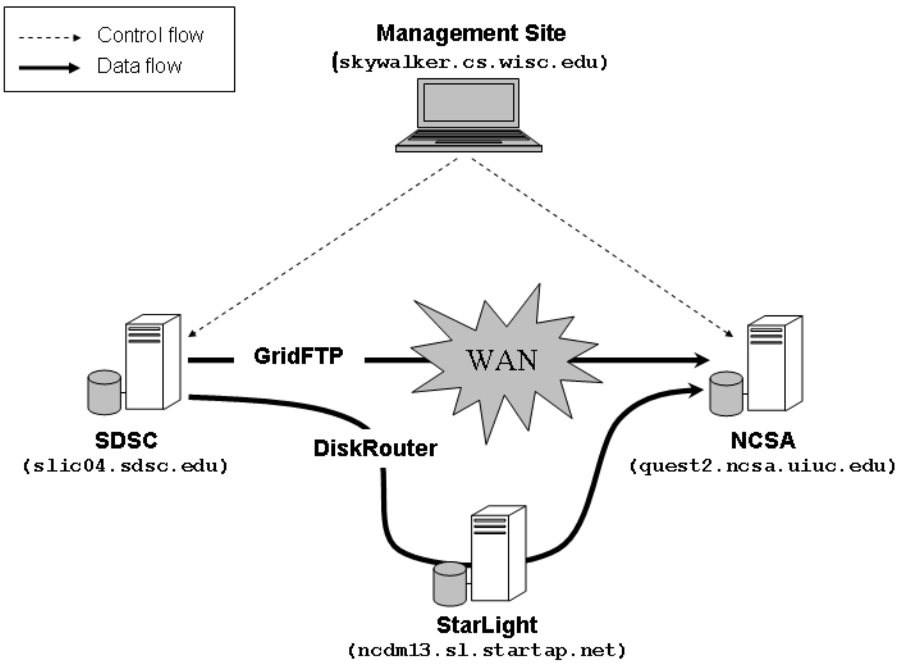

FIG. 5.1. *Experiment Setup. DiskRouter and GridFTP protocols are used to transfer data from SDSC to NCSA. Stork was running at the Management site,a nd making scheduling decisions for the transfers.*

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 5.2. 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.

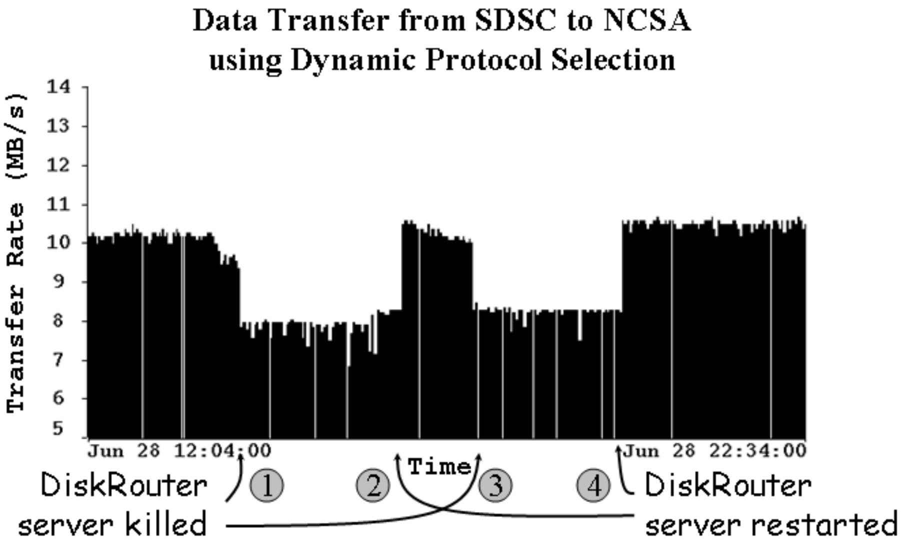

Fig. 5.2. *Dynamic Protocol Selection. The DiskRouter server running on the SDSC machine gets killed twice at points (1) and (3), and it gets restarted at points (2) and (4). In both cases, Stork employed next available protocol (GridFTP in this case) to complete the transfers.*

5.2. Experiment 2: Testing the Run-time Protocol Auto-tuning. In the second experiment, we submitted another 500 data transfer requests to the Stork server. Each request was to transfer a 1.1GB image file (total 550 GB) using GridFTP as the primary protocol. We used third-party globus-url-copy transfers without any tuning and without changing any of the default parameters.

Parameter	Before auto-tuning	After auto-tuning
parallelism	1 TCP stream	4 TCP streams
block size	1 MB	1 MB
tcp buffer size	64 KB	256 KB

Table 5.1
Network parameters for gridFTP before and after auto-tuning feature of Stork being turned on.

We turned off the auto-tuning feature of Stork at the beginning of the experiment intentionally. The average data transfer rate that globus-url-copy could get without any tuning was only 0.5 MB/s. The default network parameters used by globus-url-copy are shown in Table 1. After a while, we turned on the auto-tuning feature of Stork. Stork first obtained the optimal values for I/O block size, TCP buffer size and the number of parallel TCP streams from the monitoring and tuning infrastructure. Then it applied these values to the subsequent transfers. Figure 5.3 shows the increase in the performance after the auto-tuning feature is turned on. We got a speedup of close to 20 times compared to transfers without tuning.

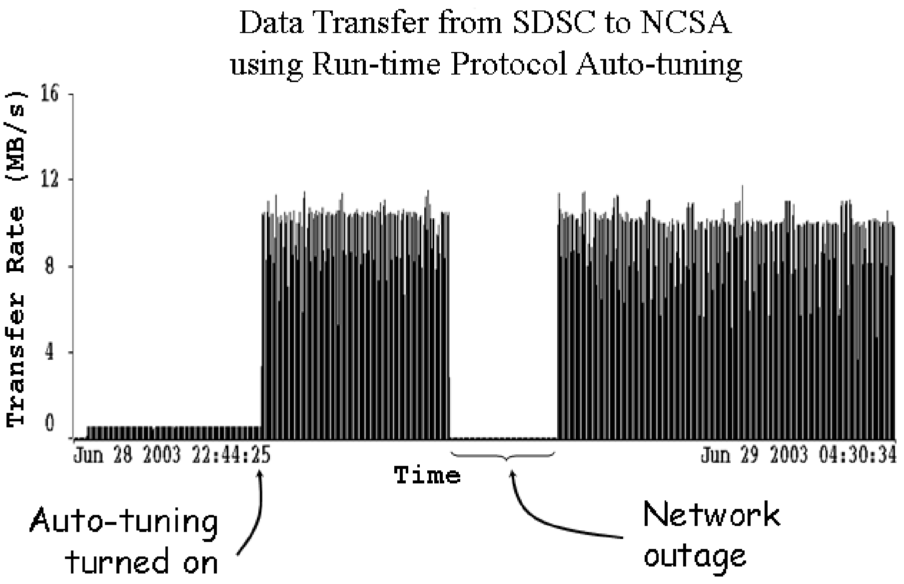

FIG. 5.3. *Run-time Protocol Auto-tuning. Stork starts the transfers using the GridFTP protocol with auto-tuning turned off intentionally. Then we turn the auto-tuning on, and the performance increases drastically.*

6. Future Work. We are planning to enhance the dynamic protocol selection feature of Stork, so that it will not only select any available protocol to perform the transfer, but it will select the best one. The requirements of 'being the best protocol' may vary from user to user. Some users may be interested in better performance, and others in better security or better reliability. Even the definition of 'better performance' may vary from user to user. We are looking into the semantics of how to to define 'the best' according to each user's requirements.

We are also planning to add a feature to Stork to dynamically select which route to use in the transfers and then dynamically deploy DiskRouters at the nodes on that route. This will enable us to use the optimal routes in the transfers, as well as optimal use of the available bandwidth throughout that route.

7. Conclusion. In this paper, we have shown a method to dynamically adapt data placement jobs to the environment at the execution time. We have 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. With two experiments, 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.

Acknowledgements. We would like to thank Robert J. Brunner, Michelle Butler and Jason Alt from NCSA; Philip Papadopoulos, Mason J. Katz and George Kremenek from SDSC for the invaluable help in providing us access to their resources, support and feedback.

REFERENCES

[1] B. Allcock, J. Bester, J. Bresnahan, A. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel and S. Tuecke, *Secure, Efficient Data Transport and Replica Management for High-Performance Data-Intensive Computing*, in Proceedings of IEEE Mass Storage Conference", April 2001, San Diego, California.

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

[3] J. Bent, V. Venkataramani, N. LeRoy, A. Roy, J. Stanley, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau and M. Livny, *Flexibility, Manageability, and Performance in a Grid Storage Appliance*, in Proceedings of the Eleventh IEEE Symposium on High Performance Distributed Computing (HPDC11),July 2002, Edinburgh, Scotland.

[4] 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.

[5] 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.

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

[7] C. Dovrolis, P. Ramanathan and D. Moore, *What do packet dispersion techniques measure?*, in Proceedings of INFOCOMM, 2001.

[8] 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, November 1999, Portland, Oregon.

[9] M. Fisk and W. Weng, *Dynamic Right-Sizing in TCP*, in Proceedings of ICCCN, 2001.

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

[11] D. Koester, em Demonstrating the TeraGrid - A Distributed Supercomputer Machine Room, The Edge, The MITRE Advanced Technology Newsletter, (2) 2002.

[12] 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, Computer Sciences Department, 2003.

[13] T. Kosar and M. Livny, *Scheduling Data Placement Activities in the Grid*, Technical Report CS-TR-2003-1483, University of Wisconsin, Computer Sciences Department, 2003.

[14] 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, (1988), pp. 104–111.

[15] NLANR/DAST, *Auto Tuning Enabled FTP Client And Server: Autobuf*, http://dast.nlanr.net/Projects/Autobuf, 2003.

[16] NLANR/DAST, *Iperf: The TCP/UDP Bandwidth Measurement Tool*, http://dast.nlanr.net/Projects/Iperf/, 2003.

[17] 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 (CCGrid), May 2003, Tokyo, Japan.

[18] 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), July 1998, Chicago, Illinois.

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

[20] J. Semke, J. Mahdavi and M. Mathis, *Automatic TCP Buffer Tuning*, in Proceedings of SIGCOMM, pp. 315–323,1998.

[21] D. Thain and and M. Livny, *The Ethernet Approach to Grid Computing*, in Proceedings of the Twelfth IEEE Symposium on High Performance Distributed Computing (HPDC12), June 2003, Seattle, Washington.

[22] D. Thain, J. Basney and 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 (HPDC10), August 2001, San Francisco, California.

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

[24] 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.

[25] 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), August 1996, Portland, Oregon.