The Kangaroo Approach to Data Movement on the Grid

Douglas Thain, Jim Basney, Se-Chang Son, and Miron Livny
Computer Sciences Department,
University of Wisconsin-Madison
{thain,jbasney,sschang,miron}@cs.wisc.edu

Abstract

Access to remote data is one of the principal challenges of grid computing. While performing I/O, grid applications must be prepared for server crashes, performance variations, and exhausted resources. To achieve high throughput in such a hostile environment, applications need a resilient service that moves data while hiding errors and latencies. We illustrate this idea with Kangaroo, a simple data movement system that makes opportunistic use of disks and networks to keep applications running. We demonstrate that Kangaroo can achieve better end-to-end performance than traditional data movement techniques, even though its individual components do not achieve high performance.

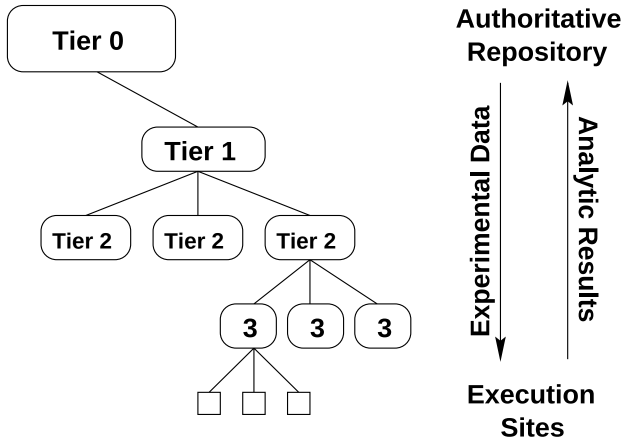


Figure 1. Hierarchical Data Grid

1 Introduction

Grid computing introduces a host of problems into the matter of attaching an application to its storage. Distributed systems are prone to performance variations, failed connections, and exhausted resources. These problems cannot be solved merely by increasing hardware capacity or reliability. They are often integral properties of distributed hardware [6], opportunistic resources [21], and social scheduling constraints.

Grid applications are not prepared to deal with any of these conditions. Often designed to run in the relatively predictable environment of a standalone machine, they expect low latency, reliable delivery, and unlimited storage. They don't schedule I/O operations or recover gracefully from unexpected failures.

We can solve these problems by re-using an old idea [14]. Traditional operating systems deal with the vagaries of disks by making a background process responsible for scheduling, coalescing, and retrying operations. Applications are not bothered with seek delays, damaged blocks, or spin-up times. As a pleasant side effect, throughput is increased by performing I/O and CPU tasks simultaneously.

The same principle can be applied to grid computing.

In this paper, we illustrate a data-movement system called Kangaroo. Kangaroo improves the reliability and throughput of grid applications by hiding network storage devices behind memory and disk buffers. Background processes are made responsible for moving data and handling errors. Applications perceive Kangaroo to be a mere file system and need not be re-written or re-compiled to become grid-aware. Kangaroo is user-level software that does not require special permissions to install or use.

Kangaroo offers a highly-available and highly-reliable service by sacrificing some consistency guarantees. Although this would be unacceptable for a general-purpose local file system, it is sensible for distributed data analysis. Major grid data efforts [3, 9, 15] note that many scientific data sets are created once and then remain read only. Organizations such as the Grid Physics Network [1] emphasize the use of hierarchical facilities for accessing large data sets, as shown in Figure 1. In such a system, experimentally-produced data flows from a central repository toward the leaves, while results computed from data move in the opposite direction. In such systems, read/write consistency is not a problem. Availability, reliability, and throughput are the main concerns.

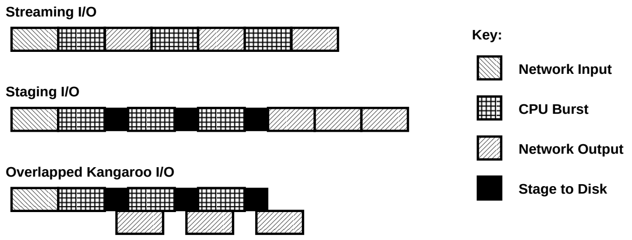


Figure 2. I/O Models

Kangaroo seeks to improve total application performance by making good use of limited resources. However, if viewed through a narrow lens, individual components of Kangaroo are clearly not high performance. We intend to make up any small-scale losses by using multiple resources at once.

An example of this principle is shown in Figure 2, which gives a time line for an application using three different I/O models – streaming, staging, and Kangaroo. A *streaming* application performs blocking I/O directly over the network while it executes. A *staging* application performs I/O on a local buffer, and then performs a blocking write of all dirty data after execution completes. In the Kangaroo I/O model, write bursts are written to a buffer and then performed concurrently with CPU bursts. As we will show below, the exact performance of the I/O bursts doesn't matter – an overall speedup is gained by using the CPU and performing I/O at the same time.

2 Design Principles

Kangaroo draws ideas from many previous works, but differs in its goals and assumptions. Before embarking upon on a description of Kangaroo, we would like to lay out the principles that have guided its design.

1. *Keep it simple.* [20]

2. *Use all available resources to hide latency.* Applications rarely use all available resources to capacity. If one resource is a bottleneck then other excess resources can be used to satisfy the demand. In practice, this means using memory and disk to handle overflow network traffic.

3. *Stop errors from reaching applications.* Scientific applications respond to errors such as "host not found" or "connection lost" by crashing or simply terminating. Delivering such errors produces no useful results. A data movement system should squash such errors by retrying, delaying, or reporting the error to a scheduler or a human operator.

4. *Sacrifice consistency for availability.* Many Grid applications are not concerned with read/write consistency. Those that are must manage a larger problem involving multiple storage sites and administrative domains. Kangaroo is only a part of this picture. We will provide an interface sufficient to manage consistency, but not to enforce it in all cases. We note that other popular file systems, such as NFS [23] and AFS [16] have bent the rules of Unix consistency with considerable success.

5. *Consider output first.* Managing inputs is harder than managing outputs. Output needs can be delayed arbitrarily, but input needs can only be anticipated using explicit information or accurate speculation. In this paper, we have concentrated upon the problem of output while maintaining a trivial system for input. With these mechanisms in place, we plan to address the problem of input in the future.

3 Architecture

The Kangaroo architecture is centered around a chainable series of servers that implement a simple interface, shown in Figure 3.

The native interface to Kangaroo is shown in Figure 3. `get` and `put` are stateless read and write operations that operate on a particular location in a target file. `get` causes the client to block until the necessary data are retrieved. `put` is a non-blocking message with no response. `commit` causes the caller to block until all outstanding `puts` have been accepted for delivery. `push` causes the caller to block until all outstanding `puts` have been transferred to their ultimate destination.

Each call includes an explicit reference to the host at which the primary data copy is stored. This (host,file) combination serves as a system-wide unique name for a data object. A Kangaroo system may service requests for this object from many different replicas, but the client need not know of or refer to such copies. The client may communicate with any server – preferably the closest – to accomplish I/O on any object.

With these four calls, we may implement a simple file service with a single server process. This is called *direct* Kangaroo and is shown in Figure 4. A client makes a TCP connection to the server to perform `gets` and `puts` on the files that it needs. The server simply executes the operations on the attached file system. This configuration is similar in form, reliability, and performance to RPC-based systems such as NFS [23].

The next step in complexity is *one-hop* Kangaroo, shown in Figure 5. Here, a second server is placed at the execution site. It satisfies `put` requests by immediately spooling them

Figure 3. Client Interface

```
int  kangaroo_get    (host,path,offset,length,data)
void kangaroo_put    (host,path,offset,length,data)
int  kangaroo_commit ()
int  kangaroo_push   (host,path)
```

Figure 4. Direct Kangaroo

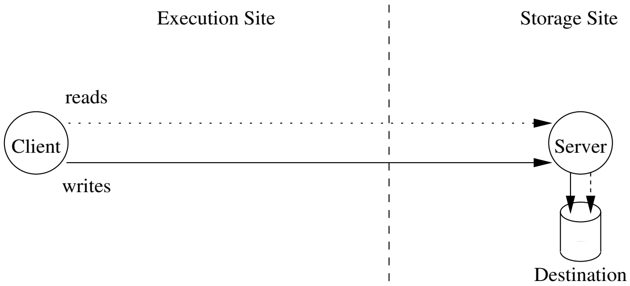


Figure 5. One Hop Kangaroo

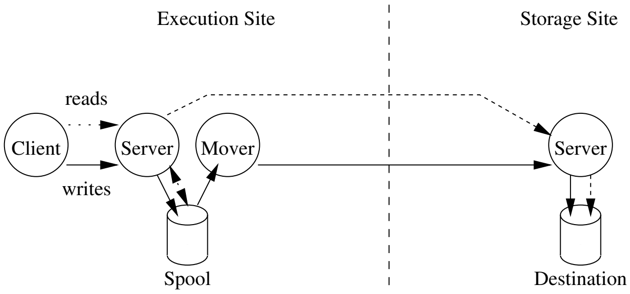


Figure 6. Two Hop Kangaroo

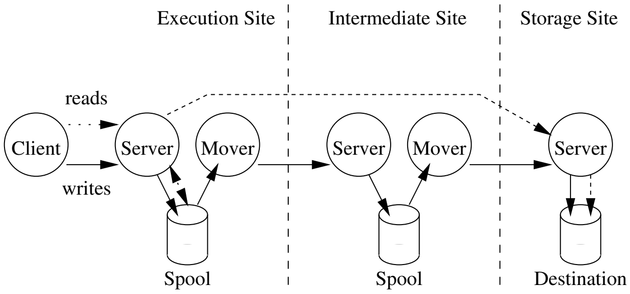
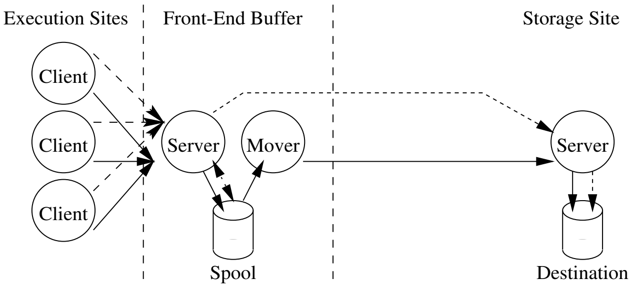


Figure 7. "Escaping" Remote Storage



to disk. A background process, the *mover*, is responsible for reading these requests and forwarding them to the destination as the network permits. `get` requests are satisfied by first consulting the local spool. If the data are not present, then `get` is invoked on the destination server.

One-hop Kangaroo insulates the client from many difficulties. If the network should fail or the destination machine should crash, the client will still be able to write to the local spool disk. Likewise, if traffic or scheduling concerns prevent the application from getting the necessary output bandwidth, it will be able to run at full speed while the mover does its job. Read operations may be satisfied from cached data without contacting the destination server.

More hops may be added, as demonstrated by *two-hop* Kangaroo in Figure 6. A multi-hop Kangaroo system can provide a number of benefits.

Multiple hops allow transfers over many network segments to be performed incrementally, avoiding the need to co-allocate network resources along all hops. This can be particularly useful for transfers over links with significant performance variations or outages. Without intermediate buffering, the performance of end-to-end connections is determined by the slowest link at any given time and end-to-end reliability is determined by the product of the up-times of the individual links.

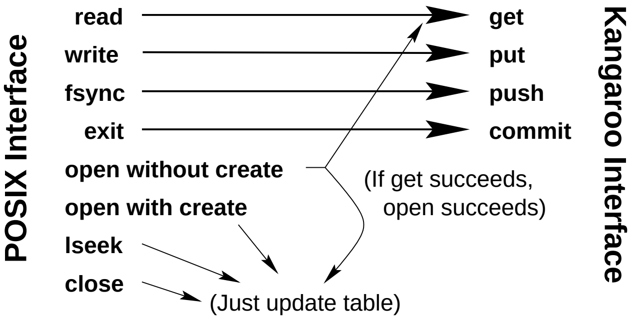
Multiple hops can also increase the available spooling space. Kangaroo can only hide network latencies if it has space to store all extant data. If a spool disk fills, the application's I/O will be reduced to end-to-end network speeds, because the application will be able to insert new data only as fast as Kangaroo can move it out. When local spool space becomes full, a server can offload spooled blocks in order to avoid slowing down the application.

Multiple servers can be used to free certain resources a quickly as possible, as shown in Figure 7. In many batch scheduling scenarios, the user is charged for occupying any resource on a compute node. To avoid holding on to a node longer than necessary, Kangaroo can be used to 'escape' the execution sites by offloading all data to a nearby storage resource. The data can then be transferred over the wide-area network as conditions permit.

3.1 Interface

The Kangaroo interface is very simple, and application writers might choose to use it directly. However, given the wide variety of available storage systems and the number of extant grid applications, it is unreasonable to expect programmers to convert existing applications to work with Kangaroo or any other newcomer to distributed systems.

To ease such transitions, we have built, using Bypass [24], an adaptation layer that converts standard POSIX operations into Kangaroo operations. This adaptation layer



fd	kind	host	path	offset
0	Unix		/dev/null	0
1	Kang	coral	/tmp/out.2	1056
2	Kang	coral	/tmp/err.2	3122
3	Unix		/etc/hosts	785
4	Kang	dbhost	/data/db	59687
...	...	...	...	...

Figure 8. Adaptation Layer

can be transparently applied to any dynamically linked program without special privileges. The layer does not affect the operation of normal filenames, but transparently 'mounts' Kangaroo into the root filesystem. Operations on filenames such as `/kangaroo/host/path` are transparently converted into Kangaroo client operations. Because Kangaroo `gets` and `puts` are stateless, the adaptation layer must remember such process-specific state such as file descriptors and seek pointers. Figure 8 illustrates how these transformations take place.

In addition, we have provided command-line utilities that invoke the client library to get, put, and push whole files between Kangaroo and local files or pipes. This provides a simple method of attaching input and output streams to Kangaroo when the use of Bypass is not desired.

Because Kangaroo is intended as a drop-in replacement for a file system, it is important that it provide sufficient operations for applications to work. However, it is not currently (and perhaps will never be) a full-featured distributed file system. Like a tape or a terminal, Kangaroo presents a file-like interface without all of the trappings of a real file system.

A number of operations are missing from the Kangaroo interface. For example, there is no support for deleting files, checking access permissions, or retrieving meta-data. The adaptation layer has several strategies for dealing with applications that request these features. For most unsupported operations, it can be plainly honest: an attempt to delete a file will return the error “operation not supported”. Some operations can be converted: a small `get` serves to satisfy a check for read permissions. Other operations must simply return dummy values. Although this practice prevents the application of some standard system tools such as *ls* or *make*, it is sufficient to admit a large number of grid applications that simply must read and write data. We may add further operations to the interface as applications require.

3.2 Consistency

Like a local file system, Kangaroo maintains read/write consistency for applications using the same first-hop server. For every data block spooled for writing, a server maintains an entry in memory. Incoming `gets` first examine this data structure and attempt to satisfy the operation locally before requesting data from another server. If data can be served entirely from the local copy, no contact is made with any other server.

Kangaroo does not enforce consistency between applications at multiple sites. Applications that need consistency guarantees must explicitly synchronize using the primitives `commit` and `push`. The former is used to make data safe from crashes, while the latter is used to make changes visible to others.

`commit` causes the caller to block until all outstanding changes have been written to *some* stable storage. This does not mean the changes are visible to all other callers! In practice, `commit` causes the receiving server to flush all buffered data and all file systems to disk. An application that `commits` may safely exit knowing that its results will eventually flow back to the destination, even if some intervening links or servers fail.

`push` causes the caller to block until all outstanding changes have been delivered to their respective destinations. In practice, `push` causes the receiver server to block until the mover has drained all dirty blocks to the next server in line. Then, the `push` is recursively called on the next server. At the target host, `push` succeeds when all outstanding data are committed into the local file system. The success message is then passed back, step-by-step, to the caller. Of course, any of these links may fail due to network or server problems. In each case, a push is free to retry the error or return it the caller. An error return does not mean the delivery has failed, but rather that the system cannot determine if the data have yet arrived. The responsibility of retrying until success lies with the top-level caller.

The adaptation layer converts POSIX operations into the appropriate Kangaroo consistency operations. When a program exits, the adaptation layer forces a `commit` to the local Kangaroo server. This prevents the system from entering a state where a program reports successful completion but loses its output to a subsequent server crash. If the user (or scheduler) that started the job wishes to wait until all data arrives, then a manual `push` should be issued. During execution, a POSIX `fsync` is also converted into a `push`. This allows existing applications that synchronize with `fsync` to operate correctly with Kangaroo.

Because output data may be arbitrarily delayed – even beyond the end of the program – `puts` are not allowed to fail and thus return no value. If a temporary resource limit, such as a full disk, prevents a server from accepting a `put`, it is free to block the caller simply by not consuming any more data from the connection. If some other error prevents committing data to the target file system, for example, insufficient privilege, then data may be stored in a local buffer. In this case, the server should contact the user to rectify the problem. A `commit` will succeed on data buffered for an 'unsolvable' problem, but a `push` will not.

As the mover process flows data in the background, it uses the same primitives as any other client of the system. As it reads dirty blocks out of the local spool, it performs `puts` on the target server. Blocks are not deleted out of the local spool until the mover successfully performs a `commit` on the target.

Any catastrophic errors must be communicated back to the scheduling system. For example, if a server crashes or suffers an unrecoverable error, the process at the other end of the connection will be abrubtly disconnected. If it can, the process should roll back to the last `commit`. For the mover, this is easy – it simply throws out its list of sent blocks and starts over. For an application, things are more complicated. An application written to the native Kangaroo interface should be desined to either roll back or abort. If using the adapation layer, a process is forcibly killed when the connection is lost. This action must be understood by the CPU scheduling system to indicate a rollback. In the case of Condor [21], a killed process is restarted from the beginning or from the last checkpoint, if available.

3.3 Scheduling

Although this architecture has been primarily cast as an on-demand data movement system, it has a natural method of integrating with a network scheduler. The mover process is implemented with Cedar, a general-purpose network socket library that supports bandwidth allocation. When establishing a new connection, the library first requests a network allocation from a site network manager. The network manager allocates bandwidth fairly among active Kangaroo connections without exceeding maximum rates configured by an administrator. Periodically, the network manager requests reports from all clients and re-allocates the bandwidth based on recent usage. At our site, this is used to enforce an upper limit on network resources consumed by opportunistically scheduled jobs.

4 Implementation

4.1 Status

We have built a Kangaroo prototype that implements the architecture described above. The basic architecture leaves a number of things unspecified to the implementation. Currently, these are:

1. *Caching discipline.* Because files are assumed to be write-once, a server is free to cache any data that passes through it. Currently, no caching is done. All `get` operations read through to the destination server. We will address this in a future work.
2. *Server discovery.* A client is free to use any server it can locate. Naturally, it has a vested interest in finding the closest one. Currently, the client library consults an environment variable for the name of the closest server and falls back on the local host.
3. *Routing mechanism.* A wide variety of route-finding protcols and mechanisms are available for computer networks. Currently, each server is equipped with a static routing table. This has not proven to be a burden, as the default behavior is to route all operations directly to the server named in the request. All one-hop configurations work without any manual routing configuration.
4. *Authentication.* Two authentication mechanisms are currently implemented: address-based and Globus GSI [12]. Each server runs as a non-privileged user and decides when a connection is made whether to trust all incoming operations.
5. *Management Tools.* To allow the user to locate data in transit and diagnose problems in the system, additional tools allow the user to query the contents of each spool directory and retrieve messages detailing failure (or success) of delivery. We envision that the server will eventually report problems to the user, instead of making the user manually query.

4.2 Performance

We evaluated our prototype in three aspects: reliability, burst performance, and overlap performance. Briefly, we confirmed that the prototype provides improved application throughput, even though individual components are not high performance. All experiments were performed on commodity workstations running Linux 2.2.17 with 512 MB of memory, a 25 MB/s disk/adapter combination, and a 100 Mb/s switched ethernet.

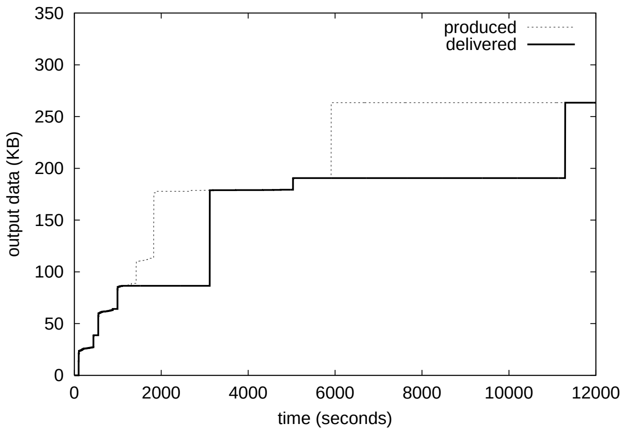


Figure 9. Response to Failures

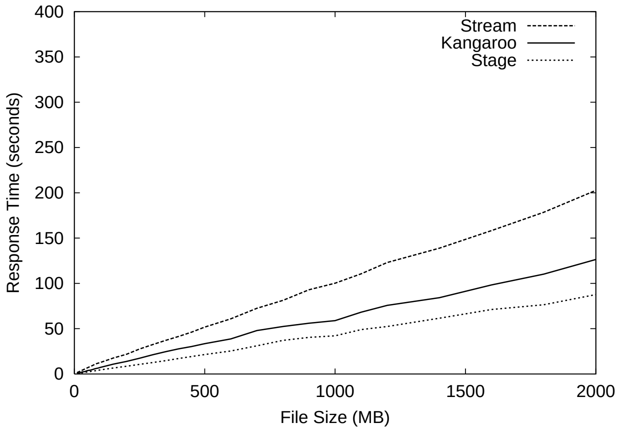


Figure 10. Burst Response Time

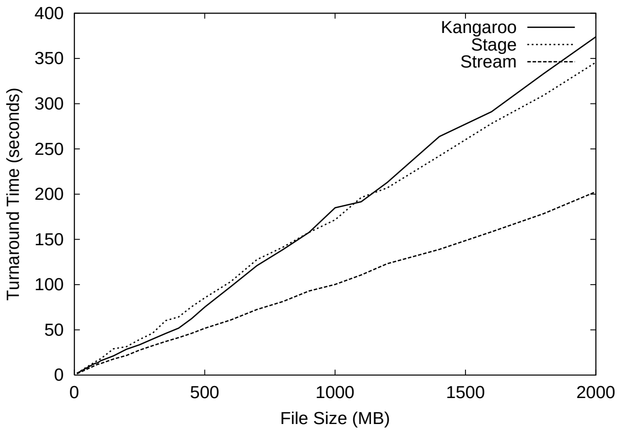


Figure 11. Burst Turnaround Time

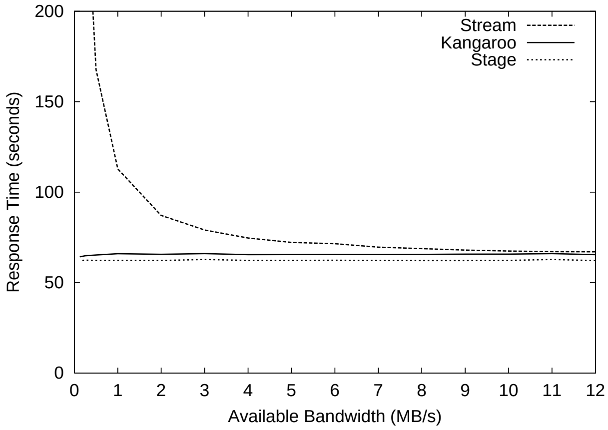


Figure 12. Images Response Time

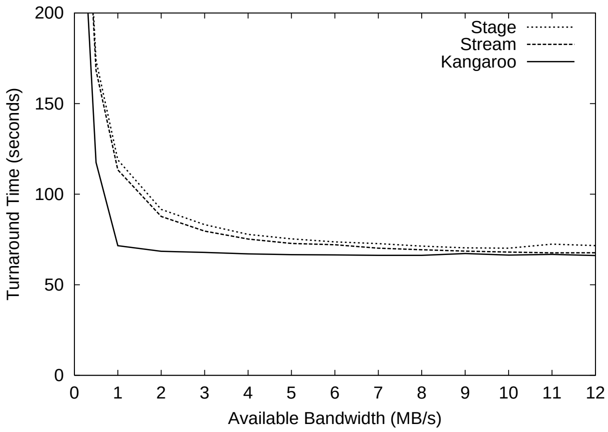


Figure 13. Images Turnaround Time

To demonstrate reliability, we used a one-hop Kangaroo system to deliver an output file generated by Gaussian [13], a popular chemistry application. Gaussian has very modest I/O needs. A typical run reads a small input file and produces a log of megabytes to gigabytes over the course of hours or days. A large amount of temporary storage is used during the run, but is not relevant to the final result.

Figure 9 shows the output behavior of a typical Gaussian run with its log file delivered through a one-hop Kangaroo system. The thin line shows the total amount of data produced by the application. The thick line shows the total amount of data actually delivered to the storage site. Service interruptions were created at 1000 and 5750 seconds by forcibly killing the destination server. As shown, an interruption simply causes data to queue up at the execution site until conditions permit output to continue. The program successfully terminated at about 6000 seconds, but service was not restored until about 11000 seconds, whereupon the output was delivered.

Of course, the improved reliability given by Kangaroo must come at a price. To quantify the cost of using the intermediate spool disk, we compared the performance of a one-hop Kangaroo system against two traditional grid transfer techniques: streaming and staging. A *streaming* application application writes its output directly to the network without any intervening storage. A *staging* application writes its whole output to the local disk and then transfers the whole file at the end of execution.

To compare the three, we created a program which generated a single burst of data as fast as possible. We measured the response and turnaround time for the generation of 10-2000 MB of data. The former was defined simply as the interval of execution, while the latter included the execution time plus any additional time to move the output to its destination.

The results of this comparison are shown in Figures 10 and 11. By definition, streaming has the same response and turnaround time. Staging gives better response time by using the faster disk during execution, but takes longer to eventually deliver the output. One-hop Kangaroo fits somewhere in between. For all files, it provides response time between staging and streaming. For files that fit in memory, it gives better turnaround time than staging. For files larger than memory, it is slightly slower.

The real benefit from Kangaroo comes from its ability to overlap CPU and I/O intervals. To demonstrate the potential benefits of overlap, we constructed a synthetic image-processing application with moderate output needs. This application approximates a number of scientific applications that produce multiple derivative data sets from an original. The application reads a single image of 5.5 MB and then produces ten output images of the same size, each a slightly different enhancement of the original. Each output required 6.1 seconds of cpu time to generate.

This application was run in three different configurations as in the previous experiment. These correspond to the three models shown in figure 2. The benefit to be gained from overlapping depends heavily on the actual ratio of CPU time to I/O time. To vary this ratio, we artificially restricted the I/O bandwidth accepted by the destination server.

Figure 12 shows the response time for this application. As expected, the streaming variant is controlled solely by the available network bandwidth. The response time for the staging and Kangaroo variants is constant, Kangaroo only slightly slower.

Figure 13 shows that Kangaroo provides a better turnaround time in all cases to its ability to overlap CPU with I/O. The turnaround time with Kangaroo remains almost constant until the available bandwidth begins to fall below the application's true I/O needs: about 1 MB/s. The other I/O disciplines are sensitive to available bandwidth in every region, even when CPU requirements are the major bottleneck.

5 Related Work

Our work is indebted to a large body of research on file systems, but we must emphasize that the usual formulation of a file system as a kernel-provided resource is not suitable for grid computing. No single file system meets the needs of users and administrators everywhere, and visiting applications do not have the permissions necessary to install privileged software.

To combat this, a grid application must bring along its own I/O system and a method for attaching to it. Several mechanisms have been proposed. Kangaroo uses library preloading, facilitated by Bypass [24]. Many other mechanisms are possible, including system call interception [2, 18], static relinking [22], binary rewriting [27, 17] and emulation through an existing interface [26].

Using these mechanisms, a variety of data-movement systems may be attached. Representative examples include Condor [22], GASS [7], and Legion [26]. The Condor remote system call facility performs all application I/O as fine-grained read and write operations over a TCP connection to the submission site. GASS allows an application to pull and push whole files synchronously when they are opened and closed, respectively. GASS also allows files opened for appending to stream data directly over a TCP connection. Legion provides a 'legacy' interface similar to GASS and a 'native' interface similar to Condor.

None of these systems address the issues of reliability or latency hiding. A Condor job that is disconnected will be immediately killed and rolled back to the last checkpoint. Failed operations in GASS and Legion result in a error propagated to the application. All of these systems cause the caller to be blocked while I/O operations are performed synchronously.

Reliability has been explored by several kernel-level file systems, such as Coda [19] and Echo [8]. These systems are very concerned with maintaining wide-area consistency, but make 'optimistic' assumptions when the network is not available. Latency hiding has been explored with the concept of buffer servers [5], particularly in the use of microproxies [4] to intercept and buffer NFS operations.

It has been suggested that Kangaroo bears a certain similarity to peer-to-peer file-sharing systems, such as Gnutella or Freenet [10]. Although the interface is similar – a client may perform I/O from any node in a cloud of Kangaroo servers – the naming is not. Kangaroo relies on the authority of a central server to provide a file's canonical name and data. Although a lack of interest may cause data to eventually be flushed from Kangaroo's distributed caches, the decision to keep or a delete a primary copy rests with the central repository.

6 Conclusion

There remain a number of avenues to explore with Kangaroo.

Foremost, we have not yet address the matter of making input data arrive exactly when it is needed. Other work [11] has suggested that bandwidth-limited prefetching is a useful model. In a large data grid, there may be multiple servers from which a read cache miss may be satisfied. Kangaroo could be coupled with a replica management system [3, 25] in order to find the 'best' replica to bring into the cache.

Currently, the application, server, and mover rely on the local operating system to mediate their demands for memory, disk, and network resources. This may not always provide for optimal end-to-end throughput. Allocating resources to the server improves the application's short-term latency, but allocating resources to the mover reduces the total storage consumed. A more informed allocation system is needed.

Finally, our current implementation authenticates client-server and server-server connections using the Globus tools [12]. This is only suitable when one person owns all of the participating servers. We wish to investigate techniques that sign individual data items, thus allowing the participating servers to be shared among multiple users.

In this paper, we have shown that a simple, unoptimized system can improve the reliability and throughput of grid applications. We have emphasized that Kangaroo offers higher throughput through flexible use of available resources, even though individual components are not high performance.

References

[1] The Grid Physics Network (GriPhyN). http://www.griphyn.org, June 2001.

[2] A. Alexandrov, M. Ibel, K. Schauser, and C. Scheiman. UFO: A personal global file system based on user-level extensions to the operating system. *ACM Transactions on Computer Systems*, pages 207–233, August 1998.

[3] 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. submitted for publication.

[4] D. Anderson, J. Chase, and A. Vahdat. Interposed request routing for scalable network storage. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, 2000.

[5] D. Anderson, K. Yocum, and J. Chase. A case for buffer servers. In *Proceedings of the IEEE Workshop on Hot Topics on Operating Systems*, April 1999.

[6] R. H. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. E. Culler, J. M. Hellerstein, D. Patterson, and K. Yelick. Cluster I/O with River: Making the fast case common. In *Proceedings of IOPADS*, May 1999.

[7] J. Bester, I. Foster, C. Kesselman, J. Tedesco, and S. Tuecke. GASS: A Data Movement and Access Service for Wide Area Computing Systems. In *Proceedings of the Sixth Workshop on I/O in Parallel and Distributed Systems*, May 1999.

[8] A. D. Birrell, A. Hisgen, C. Jerian, T. Mann, and G. Swart. The Echo distributed file system. Technical Report 111, Digital Equipment Corporation, Palo Alto, CA, USA, 1993.

[9] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke. The Data Grid: Towards an architecture for the distributed management and analysis of large scientific datasets. In *Proceedings of the Network Storage Symposium*, October 1999.

[10] I. Clarke, O. Sandberg, B. Wiley, and T. Hong. Freenet: A distributed anonymous information storage and retrieval system. In H. Federrath, editor, *Designing Privacy Enhancing Technologies: International Workshop on Design Issues in Anonymity and Unobservability*, New York, 2001. Springer.

[11] M. Crovella and P. Barford. The network effects of prefetching. In *Proceedings of the IEEE INFOCOM*, February 1997.

[12] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A security architecture for computational grids. In *Proceedings of the 5th ACM Conference on Computer and Communications Security Conference*, pages 83–92, 1998.

[13] M. J. Frisch, G. W. Trucks, H. B. Schlegel, G. E. Scuseria, M. A. Robb, J. R. Cheeseman, V. G. Zakrzewski, J. J. A. Montgomery, R. E. Stratmann, J. C. Burant, S. Dapprich, J. M. Millam, A. D. Daniels, K. N. Kudin, M. C. Strain, O. Farkas, J. Tomasi, V. Barone, M. Cossi, R. Cammi, B. Mennucci, C. Pomelli, C. Adamo, S. Clifford, J. Ochterski, G. A. Petersson, P. Y. Ayala, Q. Cui, K. Morokuma, D. K. Malick, A. D. Rabuck, K. Raghavachari, J. B. Foresman, J. Cioslowski, J. V. Ortiz, A. G. Baboul, B. B. Stefanov, G. Liu, A. Liashenko, P. Piskorz, I. Komaromi,

R. Gomperts, R. L. Martin, D. J. Fox, T. Keith, M. A. Al-Laham, C. Y. Peng, A. Nanayakkara, C. Gonzalez, M. Challacombe, P. M. W. Gill, B. Johnson, W. Chen, M. W. Wong, J. L. Andres, C. Gonzalez, M. Head-Gordon, E. S. Replogle, and J. A. Pople. Gaussian 98 revision a.7, 1998.

[14] H. Hellerman and H. J. Smith. Throughput analysis of some idealized input, output, and compute overlap configurations. *ACM Computing Surveys*, 2(2), 1970.

[15] W. Hoscheck, J. Jaen-Martinex, A. Samar, H. Stockinger, and K. Stockinger. Data management in an international data grid project. In *Proceedings of the 1st IEEE/ACM International Workshop on Grid Computing*, 2000.

[16] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systmes*, 6(1):51–81, February 1988.

[17] G. Hunt and D. Brubacher. Detours: Binary interception of Win32 functions. Technical Report MSR-TR-98-33, Microsoft Research, February 1999.

[18] M. B. Jones. Interposition agents: Transparently interposing user code at the system interface. In *Proceedings of the 14th ACM symposium on operating systems principles*, pages 80–93, 1993.

[19] J. Kistler and M. Satyanarayanan. Disconnected operation the Coda file system. *Operating Systems Review*, 23(5):213–225, December 1989.

[20] B. W. Lampson. Hints for computer system design. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, volume 17, pages 33–48, 1983.

[21] M. Litzkow, M. Livny, and M. Mutka. Condor - A hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.

[22] M. J. Litzkow. Remote UNIX: Turning Idle Workstations into Cycle Servers. In *Proceedings of the 1987 Usenix Summer Conference*, pages 381–384, 1987.

[23] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network filesystem. In *Proceedings of the Summer 1985 USENIX Conference*, pages 119–130, 1985.

[24] D. Thain and M. Livny. Multiple Bypass: Interposition agents for distributed computing. *Journal of Cluster Computing*, 2:39–47, 2001.

[25] S. Vazhkudai, S. Tuecke, and I. Foster. Replica selection in the Globus data grid. In *Proceedings of the International Workshop on Data Models and Databases on Clusters and the Grid*, 2001.

[26] B. White, A. Grimshaw, and A. Nguyen-Tuong. Grid-Based File Access: The Legion I/O Model. In *Proceedings of the 9th IEEE Symposium on High Performance Distributed Systems*, August 2000.

[27] V. C. Zandy, B. P. Miller, and M. Livny. Process hijacking. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing*, 1999.