Efficient Large-scale data movement on the Grid - Augmenting the Kangaroo approach

Rajesh Rajamani and Gogul Balakrishnan
Computer Science Department,
University of Wisconsin-Madison
{raj,bgogul}@cs.wisc.edu

Abstract

Kangaroo is a wide-area data movement system that provides high-throughput data movement by overlapping CPU and I/O[1,2]. Though Kangaroo is a persistent data mover, network and/or disk failures can reduce data availability, because Kangaroo can't route the data around failures. We demonstrate that by using multiple paths to the destination, we can improve availability without significant overheads. In our improved Kangaroo, the sender is responsible for message ordering. We also use TCP's flow control mechanism to implicitly route more data along paths that offer higher bandwidth.

1. Introduction

Kangaroo[1,2] is a wide-area data movement system developed at UW-Madison. Kangaroo improves the throughput and reliability of grid applications by hiding network storage devices behind memory and disk buffers. Together with Pluggable File System (PFS)[12], Kangaroo allows unmodified applications to overlap computation with I/O. By removing the burden of data movement from the application, Kangaroo helps reduce the turnaround time of applications.

Kangaroo uses a TCP-based message-oriented protocol. Servers exchange information by passing well-defined messages to each other. The different file operations are encoded as Kangaroo messages and may contain control and data information. Kangaroo also offers a highly reliable data movement mechanism by using a write-ahead log and retransmitting messages in case of network failures or when a server downstream runs out of spool space. However, the original Kangaroo prototype (hereafter referred to as vanilla Kangaroo) uses a static single route. This route is the first match that it finds in the Kangaroo routing table. Since it uses a single route, data cannot be routed around failures, even if alternate routes exist. This can affect the availability of data at the destination. The vanilla implementation is also not able to identify operations that can be performed in parallel, which results in wasted bandwidth.

In this paper, we describe the challenges in routing data along multiple routes to the destination and our approach to solving some of them. Section 2 gives a quick overview of the original Kangaroo architecture and interface. Section 3 gives the motivation for this work and we describe our implementation in section 4. Section 5 compares the performance of our implementation (hereafter referred to as multiroute Kangaroo) with that of the vanilla implementation. We present related work in Section 6, future work in Section 7 and conclude in Section 8.

2. Architecture

The vanilla Kangaroo architecture [1,2] is centered around a chainable series of servers that implement a simple interface

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

All the above functions except *kangaroo_put* are Remote Procedure Calls (RPCs). *kangaroo_put* and *kangaroo_get* allow the servers to fetch data from any reachable host/filesystem. A host is reachable if it is running the Kangaroo server to which the caller machine can authenticate. Currently, Kangaroo supports two forms of authentication–address-based and Globus Grid Security Infrastructure (GSI) [13].

kangaroo_commit ensures that all outstanding puts have been accepted for delivery. In practice, this is achieved by returning to the caller only after ensuring that all the messages sent prior to a commit have been logged on persistent storage at the next hop. *kangaroo_push* blocks until all outstanding puts have been transferred to their ultimate destination. We can think of this as a recursive RPC, in that, each callee invokes push and returns when the server downstream returns. A *kangaroo_push* call on the destination returns when all the data has made it to the proper file. In other words, a *kangaroo_commit* guarantees that all the previously sent messages have been successfully spooled at the next hop Kangaroo server, whereas *kangaroo_push* returns only after all the messages are executed at the destination. Messages are removed from the local spool only after a successful commit/push.

3. Motivation

As mentioned briefly above, vanilla Kangaroo uses single static routes to the destination. This can be a problem if one of the Kangaroo servers on the path runs out of disk space or if there are network outages. By using multiple routes, we can improve the availability of data by routing around failures.

Vanilla Kangaroo is implemented using a queue. Messages are sent to the next hop in the order they are received. Parallelism, which may be present by way of independent operations that can be sent concurrently, is not exploited. For instance, if five applications write to five different files, all the puts can be sent with no regard to the order in which they arrive. In fact, if puts to the same file have different offsets, they maybe independent (An informal study of some target applications reveals a large number of independent operations to the same file). This leads to wasted bandwidth and could be a serious problem if the first Kangaroo server receives more messages than it can service, causing the applications to retry sending the messages later.

By combining the two approaches–multiple routes and identifying independent operations, we have tried to improve the availability. The biggest challenge in using multiple routes is dealing with dependent operations to the same file, so that they are executed at the destination only after the messages that it depends on reach the destination. In the next Section, we describe how we deal with this problem.

4. Implementation

The most important challenge in making Kangaroo use multiple routes is the ordering of Kangaroo messages at the destination. It might seem that use of sequence numbers as in TCP might solve the problem. But this makes the receiver responsible for the ordering of packets. This will need a considerable change in the protocol that is used by vanilla Kangaroo. So, we used an approach in which the sender assumes the responsibility of ordering packets as in vanilla Kangaroo and does not require changing the existing Kangaroo architecture.

The first Kangaroo server maintains the dependency information among the Kangaroo operations it receives in a dependency graph structure very similar to the one used in [4]. The dependency graph also identifies the operations that can be done in parallel at any instance of time. There is a dependency graph per destination/filename pair. There are mover threads per available route per file. The movers query the dependency graph for parallel operations and send them along possibly different routes. Once the dependency graph has run out of parallel operations or a specified time out occurs, all the movers belonging to the dependency graph perform a push operation. This ensures that all messages sent before the push has reached the destination. Once the dependency graph is notified of a successful *push* operation, it identifies those operations that were dependent on these pushed operations. These operations are now independent and can be done concurrently and the movers resume the cycle.

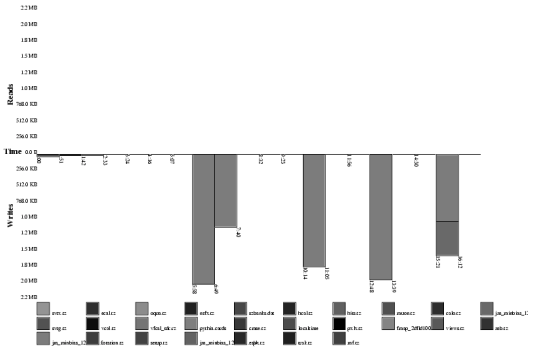
The Kangaroo servers may be required to provide different QoS guarantees for data from different simulations, to guarantee that data produced by a higher priority simulation reaches the destination first. Currently, we send as much data along a route as permitted by the TCP buffers at the Kangaroo server. If the server downstream runs out of buffer space, a backpressure is applied by TCP's flow control mechanism. This makes sure that more data is sent along other paths that have greater bandwidth. This, however, will not ensure that data produced by higher priority simulations get more bandwidth than a lower priority one. We are investigating different mechanisms to make Kangaroo servers aware of the different priorities allocated by Condor.

5. Performance

To evaluate the performance of our implementation, we first modeled the I/O needs of *cmsim*, an event simulator widely used by high-energy physicists. *cmsim*, takes as input a set of configuration files and produces anywhere between 150kB to 1MB per event.

Each simulation normally generates 500 or 1000 events for a total of 8M-1G of data over a period of 6-12 hours. We instrumented a simulation of 50 events and discovered its I/O requirements. We then wrote a model of this simulator, which produces the same amount of output at the same burst rates as the simulation. We did not model the input or CPU usage of the simulator, to keep the runtime low. This would not adversely affect our evaluation, as we were trying to measure the availability of data at the destination and not the response time of the simulations.

Normally, a batch of 100-1000 simulations use the same set of data files, so we assume that these files would be made available using a shared filesystem like AFS or NFS.



A Fig.1. The I/O rate of cmsim was instrumented. This figure shows only the output generated by the simulation.

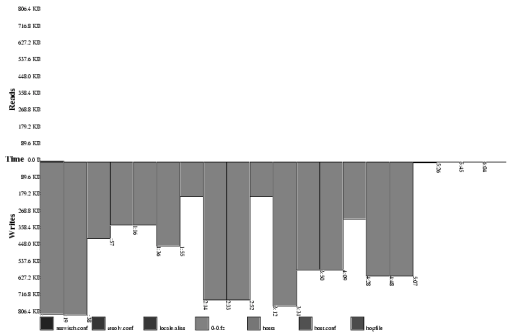


Fig.2. A model of the simulation, which produces approximately the same amount of data with similar burst rates. The computation phases were not modeled.

The output generated by cmsim is shown in Fig.1 and the output produced by our model is shown in Fig.2. The second figure shows that the data generated by our model had similar burst rates as that produced by the *cmsim* simulation. We then set up three different topologies as shown in Figures 3 through 5. We used these to measure the turnaround time of 10 simulations run sequentially. The turnaround time includes the execution time plus any additional time to move the output to its destination.

We first compare the performance of multiroute Kangaroo with that of vanilla Kangaroo for sending/fetching a file. We sent 10 files of equal size (about 8MB) sequentially to a destination five hops away. Fig.6 shows that using only a single route and the same number of resources (movers/servers), the turnaround times of the applications using multiroute Kangaroo was better than while using the vanilla implementation.

The improvement on both counts can be attributed to two facts –

- The multiroute version keeps messages in memory and doesn't have to read messages back from the disk like vanilla does for every send operation.

- Vanilla Kangaroo performs periodic commits.

Fig.7 is a comparison of the two implementations, when used to fetch 10 files sequentially from a Kangaroo server 5 hops away (Fig.3). As can be seen the performance of multiroute is much better than vanilla. In the vanilla implementation, when a get is invoked, each hop on the way to the source needs to be queried to find whether it has the required data cached. If the data is cached at any node, it means that it is in transit and has probably not made it to the destination. So, for this test each of the five hops had to be queried and since there was no data in transit, the data had to be fetched from the destination. The multiroute implementation is such that if the data is not found in the cache of the first Kangaroo server (that the application sees), the data is fetched directly from the source without having to query any of the intervening nodes. So, for this test, while vanilla had to

query five servers, multiroute had to query only two servers. By reducing the cost of fetching a file, we make it attractive for applications that require reading in files and for which using the local shared file system may not be possible.

The next test was performed on the topology shown in Fig.4, which is quite similar to what we expect to find in the Grid, i.e., a cluster where jobs are run and send the data back via two paths to the destination. We ran eighteen simulations concurrently and redirected the output to {dewey,doc}.cs.wisc.edu. While running vanilla nine simulations sent their output to dewey.cs.wisc.edu and nine others to doc.cs.wisc.edu, whereas all eighteen sent their output to both dewey.cs.wisc.edu and doc.cs.wisc.edu while running multiroute.

Vanilla Kangaroo queues all messages in FIFO order. This leads to head-of-line blocking. The model we used for testing wrote out 8MB of data, all to different offsets and then did a rename.

Fig.3. Topology 1 shows the application writing/reading from a destination five hops away.

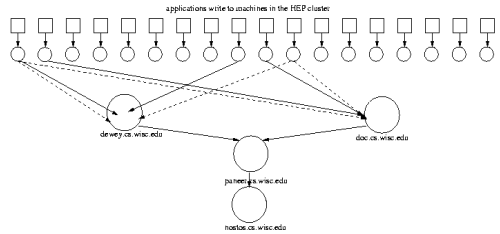


Fig.4. Topology 2 has eighteen applications writing to Kangaroo servers on the hep cluster. The data is routed through two nodes (dewey and doc), which send forward the messages to paneer, which in turn sends it the destination, nostos.cs.wisc.edu.

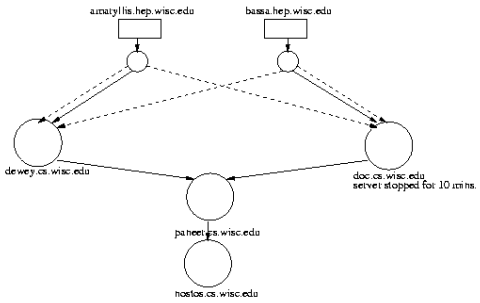


Fig.5. Topology 3, a scaled down version of topology 2, was used to test how failures affected the availability of data.

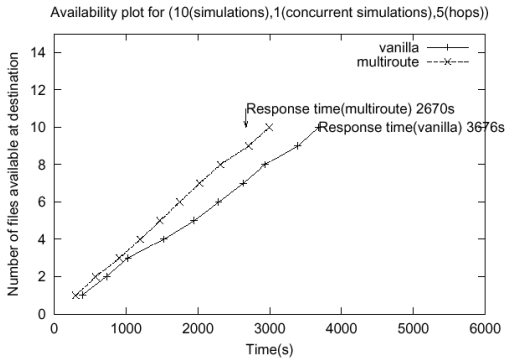


Fig.6. This figure shows the time 10 files become available at the destination, when sent one after another by an application, 5 hops away from the destination. Each file was 8642560 bytes, for a total of 86425600 bytes.

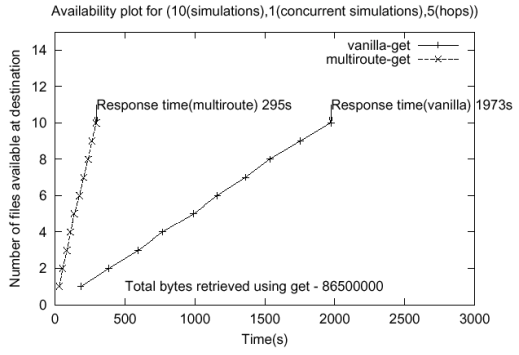


Fig.7. Shows the time taken to fetch 10 files one after another. The source was 5 hops away.

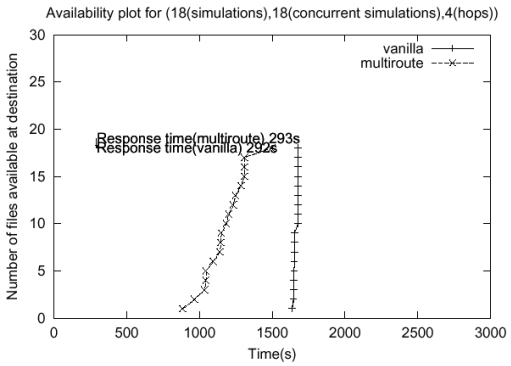


Fig.8. This illustrates the head-of-line blocking problem in the vanilla implementation. Though the data belonging to the files arrive earlier than 1650s, all the renames occur in the short interval 1600-1700s. This problem is not seen in the multiroute implementation.

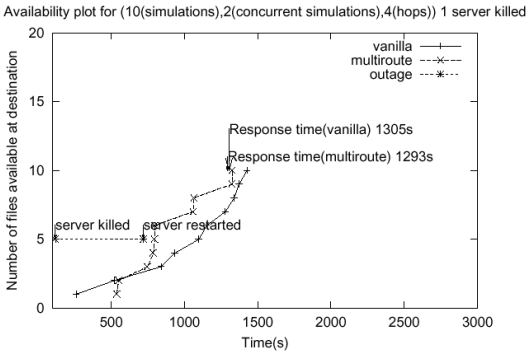


Fig.9. This figure illustrates the ability of the multiroute implementation to route data around failures.

Therefore, the only dependency was that of rename on all the puts. It must be mentioned that the *cmsim* simulator writes out a large amount of data sequentially to a file. There are no random writes to the output file. We introduced a rename at the end to introduce a dependency. The rename messages are sent after all the puts, and are inserted towards the end of the queue, which also contains the put messages of other files. This is clearly brought out in Fig.8, where all the files become available at around the same time, because the renames arrive in a short interval of time after all the data messages arrive. The graphs show that the turnaround time while using multiroute is better. This is because both inter- as well as intra-file independent operations are sent in parallel by multiroute. Quite obviously using a mover per route per file is better than using a mover per Kangaroo server. This eliminates the head-of-line blocking present in the vanilla implementation.

We performed the next test to evaluate the advantages of routing data around a failure[1]. We used a scaled down version of the topology 2, shown in Fig.5. The setup was similar, in that each node running the application sent their output to both dewey.cs.wisc.edu and doc.cs.wisc.edu while running multiroute, whereas one sent its output to dewey.cs.wisc.edu and the other to doc.cs.wisc.edu while running vanilla. We ran five simulations on each of the cluster nodes to produce a total of ten files, each of which produces approximately 8MB of data over an average of five minutes running time (The running time depends on the speed with which data can be forwarded to the Kangaroo server, which is variable). Since they take approximately the same amount of time while running on these nodes, it would help to think of this as a set of five rounds of simulations, where one file was produced per round at each of {amaryllis,bassa}.hep.wisc.edu. We stopped the Kangaroo server on doc.cs.wisc.edu for a period of ten minutes two minutes after the

[1] Data can be routed around failures only if it occurs in a physically distinct part of the topology.

simulations started. While running vanilla Kangaroo, when doc.cs.wisc.edu is shut down, only the five files being sent through dewey.cs.wisc.edu have an open path to the destination. The data generated by the simulations running on bassa.hep.wisc.edu are spooled at the local Kangaroo server at bassa.hep.wisc.edu to be sent later. When the Kangaroo server on doc.cs.wisc.edu is restarted, this spooled data is sent along. The resumption of the Kangaroo service on doc.cs.wisc.edu explains the appearance of four files between 600 and 800 seconds. The plot of turnaround times while running multiroute Kangaroo is very interesting. During the first two minutes, when the Kangaroo servers on {dewey,doc}.cs.wisc.edu were running, parts of files produced at {amaryllis,bassa}.hep.wisc.edu were sent through both of them. When doc.cs.wisc.edu is shutdown, the subsequent renames of these files don't succeed, because the 'push' along this route fails. Since, the second round of simulations start after the server on doc.cs.wisc.edu was shutdown, all data produced at {amaryllis,bassa}.hep.wisc.edu, take the route through dewey.cs.wisc.edu. So, the first file that becomes available while running multiroute is from the second round of simulations, whereas the first available file while running vanilla is the one produced at amaryllis during the first round. This explains why the first file becomes available before the first file while running multiroute. There is also a greater ramp-up in the multiroute plot after the route through doc.cs.wisc.edu is re-established. This can be explained by the greater number of movers working to move the files on both the routes. Though this is a simple test, it clearly shows the ability of multiroute Kangaroo to route data around failures. Of course, if a file has some parts of it already stored in a Kangaroo server that goes down, it doesn't become available until that Kangaroo server become available. This is the case in vanilla as well as multiroute Kangaroo. However, it is possible that more files get stuck at a node while running multiroute, if that server goes down. We feel that the additional movers that this server will use when it restarts will make sure that all the data becomes available at the destination without much greater delay than will be experienced while running vanilla.

We have concentrated on the performance improvement due to our design, which uses a mover per route per file. The aspect of scalability of Kangaroo servers has not been investigated. It would be interesting to measure the peak message processing ability of the servers and find out what the bottlenecks are. We hope to run tests to help us evaluate the servers under load.

6. Related Work

Kangaroo is an attempt to support large-scale file transfer (in the order of terabytes) services over the WAN [1, 2, 3]. Kangaroo is a persistent data mover, which improves reliability and throughput of grid applications by hiding network storage devices behind memory and disk buffers.

A network of Kangaroo servers form an overlay network, in that, the application need not be aware of the networking protocols, like say, TCP/IP, that Kangaroo uses. SMTP servers which

store and forward mails also exploit the idea of overlay networks[11]. However, the SMTP servers were not designed to handle large-scale file transfer. A similar idea is explored in [9]. In [9], a group of systems form a Resilient Overlay Network(RON) over the internet. This overlay network monitors the characteristics of the underlying internet and tries to route a packet through a RON node if the underlying internet path is not optimal. In our work we use the Kangaroo servers to hide the latency to the application and concentrate on routing around failures.

Applications that need to perform remote I/O, send the file operations to a Kangaroo server, which then is responsible for data movement and error handling. The destination may be reachable using different routes and we are trying to add a Traffic Engineering module to Kangaroo, which will send independent packets on different routes. By forwarding the packets along different paths, we expect to see an increase in throughput and link utilization. Some packets, which are large enough, can be split into smaller pieces and the smaller packets can be sent on different routes.

There are several aspects to our problem. One aspect is identifying the operations that are independent. There are several similarities between our problem of identifying independent operations and the one of identifying parallelism in ordinary programs. Most of the existing work***s consider*** creating a dependency graph or a tree of the instructions [4,5]. Such an approach has also been used in [6]. In [6], the authors try to improve the performance of the Netsolve system [7] by reducing communication overheads and scheduling operations to run in parallel whenever possible.

The other aspect is path selection. We must route the data packets so that no link is over or under utilized. The evolving field of traffic engineering concentrates on these aspects. [8] provides an architecture for balancing the traffic load on the various links, routers, and switches in the network so that none of these components is over-utilized or underutilized.

There have also been studies on the effect of path selection on the end-to-end performance of the communicating applications. In [10] the authors have studied the effect of the routing decisions on several parameter of path quality and have found that in most of the cases there is an alternate path of superior quality.

7. Future work

There are a lot of interesting aspects of the system that have to be investigated. Topmost on our list is to study the scalability characteristics of the Kangaroo servers. This would help us determine at what point using an additional route would be beneficial. The problem of route discovery remains largely unsolved. Automating the calculation of the forwarding table would be critical as the system grows. Currently, all routes are entered by hand. Given the fact that, a Kangaroo server can connect to any other Kangaroo server on the Internet, the standard routing protocols like RIP cannot be used without modifications. We would like to make Kangaroo aware of the priorities given out by Condor, a popular batch scheduling system. This would help in providing Quality-of-Service

guarantees to data generated by jobs of higher priority.

8. Conclusions

In our work, we have focused on removing certain shortcomings of the original Kangaroo implementation. We show that message ordering can be achieved without too much overhead *in certain cases* by making the sender responsible for making sure that the messages arrive in the right order at the destination. It must be mentioned that our approach would prove prohibitively costly if all or a large number of messages are dependent on previous ones. By incorporating the ability of using multiple routes, our implementation of Kangaroo can now route data around failures. Also, by exploiting the inter- and intra- file dependencies, we greatly reduce the effects of head-of-line blocking present in the original prototype. We also get rid of the excess overhead in fetching data that is not in transit, by having to do only two lookups, whereas in the original implementation the number of lookups was proportional to the number of hops to the source of the data.

References:

[1] Douglas Thain, Jim Basney, Se-chang son, Miron Livny, "The Kangaroo approach to data movement on the Grid", in Proceedings of the Tenth IEEE symposium on High Performance Distributed Computing, San Francisco, CA, Aug 2001.

[2] Douglas Thain, "An Overlay Network Architecture for Terabyte-scale Data movement", Ph.D dissertation proposal, October 2001.

[3] Douglas Thain, John Bent, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau and Miron Livny, "Gathering at the well: Creating communities for grid I/O", In Proceedings of Supercomputing 2001, Denver, Colorado, November 2001.

[4] Todd M. Austin and Gurindar S. Sohi, “Dynamic dependency analysis of ordinary programs”, in proceedings of the 19th international conference on Computer Architecture, 1992

[5] Jaime.H.Moreno and Mayan Moudgill, Scalable instruction level parallelism through tree instructions, IBM research division RC 20661 (91417)

[6] Dorian Arnold, Dieter Bachmann and Jack Dongarra, “Request Sequencing: Optimizing Communication for the Grid”, European Conference on Parallel Processing

[7] H. Casanova and J. Dongarra. “NetSolve's Network Enabled Server: Examples and Applications” IEEE Computational Science & Engineering, 5(3): 57-67, September 1998

[8] Chuck Semeria, "Traffic Engineering for the New Public Network", Web essay, http://www.juniper.net/techcenter /techpapers/200004.html.

[9] David G. Andersen, Hari Balakrishnan, M. Frans Kaashoek, Robert Morris, “Resilient Overlay Networks”, Proc. 18th ACM SOSP, Banff, Canada, October 2001

[10] Stefan Savage, Andy Collins, Eric Hoffman, John Snell, and Thomas Anderson, “The end-to-end effects of Internet path selection”, Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication, 1999

[11] RFC 821, Simple Mail Transfer Protocol, available from www.rfc-editor.org

[12] Pluggable File system (PFS). www.cs.wisc.edu/condor/pfs

[13] Globus Grid Security Infrastructure (GSI). http://www-fp.globus.org/security/