Attaching Cloud Storage to a Campus Grid Using Parrot, Chirp, and Hadoop

Patrick Donnelly, Peter Bui, Douglas Thain
Computer Science and Engineering
University of Notre Dame
pdonnel3, pbui, dthain@nd.edu

Abstract

The Hadoop filesystem is a large scale distributed filesystem used to manage and quickly process extremely large data sets. We want to utilize Hadoop to assist with data-intensive workloads in a distributed campus grid environment. Unfortunately, the Hadoop filesystem is not designed to work in such an environment easily or securely. We present a solution that bridges the Chirp distributed filesystem to Hadoop for simple access to large data sets. Chirp layers on top of Hadoop many grid computing desirables including simple deployment without special privileges, easy access via Parrot, and strong and flexible security Access Control Lists (ACL). We discuss the challenges involved in using Hadoop on a campus grid and evaluate the performance of the combined systems.

1 Introduction

A campus grid is a large distributed system encompassing all of the computing power found in a large institution, from low-end desktop computers to high end server clusters. In the last decade, a wide variety of institutions around the world have constructed campus grids by deploying software such as the Condor [16] distributed computing system. One of the largest known public campus grids is BoilerGrid, consisting of over 20,000 cores harnessed at Purdue University [2]; and systems consisting of merely a few thousand cores are found all around the country [13]. Campus grids have traditionally been most effective at running high-throughput computationally-intensive applications. However, as the data needs of applications in the campus grid increases, it becomes necessary to provide a system that can provide high capacity and scalable I/O performance.

To address this, we consider the possibility of using Hadoop [8] as a scalable filesystem to serve the I/O needs of a campus grid. Hadoop was originally designed to serve the needs of web search engines and other scalable applications that require highly scalable streaming access to large datasets. Hadoop excels at dealing with large files on the order of terabytes and petabytes while ensuring data integrity and resiliency through checksums and data replication. Hadoop is typically used to enable the massive processing of large data files through the use of distributed computing abstractions such as Map Reduce [5].

In principle, we would like to deploy Hadoop as a scalable storage service in a central location, and allow jobs running on the campus grid to access the filesystem. Unfortunately, there are a number of practical barriers to accomplishing this. Most applications are not designed to operate with the native Hadoop Java interface. When executing on a campus grid, it is not possible to install any dependent software on the local machine. The Hadoop protocol was not designed to be secure over wide area networks, and varies in significant ways across versions of the software, making if difficult to upgrade the system incrementally.

To address each of these problems, we have coupled Hadoop to the Chirp filesystem, and employed the latter to make the storage interface accessible, usable, secure, and portable. Chirp [15] is a distributed filesystem for grid computing created at Notre Dame and used around the world. It has been shown to give favorable performance [15] [14] for distributed workflows. It is primarily used to easily export a portion of the local filesystem for use by remote workers. Chirp is built to be easily deployed temporarily for a job and then torn down later. It supports unprivileged deployment, strong authentication options, sensible security of data, and grid-friendly consistency semantics. Using Parrot [12], we can transparently connect the application to Chirp and thereby to Hadoop.

The remainder of this paper presents our solution for attaching Hadoop to a campus grid via the Chirp filesystem. We begin with a detailed overview of Hadoop, Parrot, and Chirp, explaining the limitations of each. We explored two distinct solutions to this problem. By connecting Parrot directly to Hadoop, we achieve the most scalable performance, but a less robust system. By connecting Parrot to Chirp and then to Hadoop, we achieve a much more stable system, but with a modest performance penalty. The final section of the paper evaluates the scalability of our solution on both local area and campus area networks.

2 Background

The following is an overview of the systems we are trying to connect to allow campus grid users access to scalable distributed storage. First, we examine the Hadoop filesystem and provide a summary of its characteristics and limitations. Throughout the paper we refer to the Hadoop filesystem as HDFS or simply Hadoop. Next, we introduce the Chirp filesystem, which is used at the University of Notre Dame and other research sites for grid storage. Finally, we describe Parrot, a virtual filesystem adapter that provides applications transparent access to remote storage systems.

2.1 Hadoop

Hadoop [8] is an open source large scale distributed filesystem modeled after the Google File System [7]. It provides extreme scalability on commodity hardware, streaming data access, built-in replication, and active storage. Hadoop is well known for its extensive use in web search by companies such as Yahoo! as well as by researchers for data intensive tasks like genome assembly [10].

A key feature of Hadoop is its resiliency to hardware failure. The designers of Hadoop expect hardware failure to be normal during operation rather than exceptional [3]. Hadoop is a "rack-aware" filesystem that by default replicates all data blocks three times across storage components to minimize the possiblty of data loss.

Large data sets are a hallmark of the Hadoop filesystem. File sizes are anticipated to exceed hundreds of gigabytes to terabytes in size. Users may create millions of files in a single data set and still expect excellent performance. The file coherency model is "write once, read many". A file that is written and then closed cannot be written to again. In the future, Hadoop is expected to provide support for file appends. Generally, users do not ever need support for random writes to a file in everyday application.

Despite many attractive properties, Hadoop has several essential problems that make it difficult to use from a campus grid environment:

- **Interface.** Hadoop provides a native Java API for accessing the filesystem, as well as a POSIX-like C API, which is a thin wrapper around the Java API. In principle, applications could be re-written to access these interfaces directly, but this is a time consuming and intrusive change that is unattractive to most users and developers. A FUSE [1] module allows unmodified applications to access Hadoop through a user-level filesystem driver, but this method still suffers from the remaining three problems:

- **Deployment.** To access Hadoop by any method, it is necessary to install a large amount of software, including the Java virtual machine and resolve a number of dependencies between components. Using FUSE module requires administrator access to install the FUSE kernel modules, install a privileged executable, and update the local administrative group definitions to permit access. On a campus grid consisting of thousands of borrowed machines, this level of access is simply not practical.

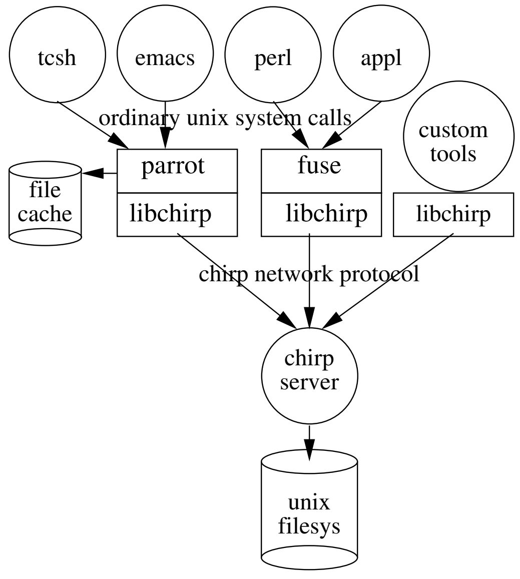
- **Authentication.** Hadoop was originally designed with the assumption that all of its components were operating on the same local area, trusted network. As a result, there is no true authentication between components – when a client connects to Hadoop, it simply claims to represent principal X, and the server component accepts that assertion at face value. On a campus-area network, this level of trust is inappropriate, and a stronger authentication method is required.

- **Interoperability.** The communication protocols used between the various components of Hadoop are also tightly coupled, and change with each release of the software as additional information is added to each interaction. This presents a problem on a campus grid with many simultaneous users, because applications may run for weeks or months at a time, requiring upgrades of both clients and servers to be done independently and incrementally. A good solution requires both forwards and backwards compatibility.

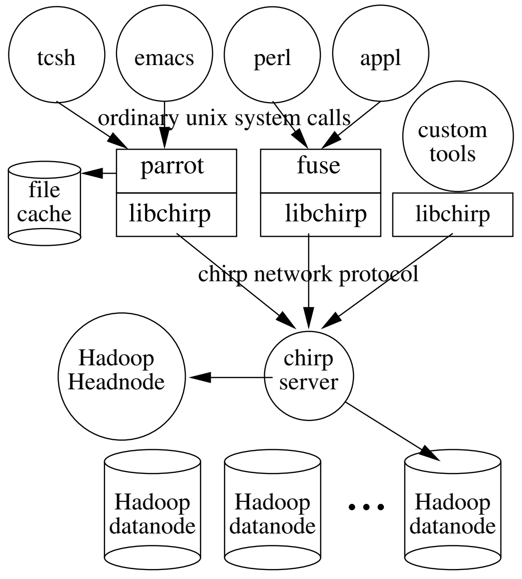
2.2 Chirp

Chirp [15] is a distributed filesystem used in grid computing for easy userlevel export and access to file data. It is a regular user level process that requires no privileged access during deployment. The user simply specifies a directory on the local filesystem to export for outside access. Chirp offers flexible and robust security through per-directory ACLs and various strong authentication modes. Figure 1a provides a general overview of the Chirp distributed filesystem and how it is used.

A Chirp server is set up using a simple command as a regular user. The server is given the root directory as an option to export to outside access. Users can authenticate to the server using various schemes including the Grid Security Infrastructure (GSI) [6], Kerberos [11], or hostnames. Access to data is controlled by a per-directory ACL system using these authentication credentials.



(a) The Chirp Filesystem



(b) The Chirp Filesystem Bridging Hadoop

Figure 1: Applications can use Parrot, FUSE, or *libchirp* directly to communicate with a Chirp server on the client side. The modifications presented in this paper, Chirp can now multiplex between a local unix filesystem and a remote storage service such as Hadoop.

A Chirp server requires neither special privileges nor kernel modification to operate. Chirp can both be deployed to a grid by a regular user to operate on personal data securely. We use Chirp at the University of Notre Dame to support workflows that require a personal file bridge to expose filesystems such as AFS [9] to grid jobs. Chirp is also used for scratch space that harnesses the extra disk space of various clusters of machines. One can also use Chirp as a cluster filesystem for greater I/O bandwidth and capacity.

2.3 Parrot

To enable existing applications to access distributed filesystems such as Chirp, the Parrot [12] virtual filesystem adapter is used to redirect I/O operations for unmodified applications. Normally, applications would need to be changed to allow them to access remote storage services such Chirp or Hadoop. With Parrot attached, unmodified applications can access remote distributed storage systems transparently. Because Parrot works by trapping program system calls through the *ptrace* debugging interface, it can be deployed and installed without any special system privileges or kernel changes.

One service Parrot supports is the Chirp filesystem previously described. Users may connect to Chirp using a variety of methods. For instance, the *libchirp* library provides an API that manages connections to a Chirp server and allows user to construct custom applications. However, a user will generally use Parrot to transparently connect existing code to a Chirp filesystem without having to interface directly with *libchirp*. While applications may also use FUSE to remotely mount a Chirp fileserver, Parrot is more practical in a campus grid system where users may not have the necessary administrative access to install and run the FUSE kernel module. Because Parrot runs in userland and has minimal dependencies, it is easy to distribute and deploy in restricted campus grid environments.

3 Design and Implementation

This section describes the changes we have made to implement a usable and robust system that allows for applications to transparently access the Hadoop filesystem. The first approach is to connect Parrot directly to HDFS, which provides for scalable performance, but has some limitations. The second approach is to connect Parrot to Chirp and thereby Hadoop, which allows us to overcome these limitations with a modest performance penalty.

3.1 Parrot + HDFS

Our initial attempt at integrating Hadoop into our campus grid environment was to utilize the Parrot middleware filesystem adapter to communicate with our test Hadoop cluster. To do this we used *libhdfs*, a C/C++ binding for HDFS provided by the Hadoop project, to implement a HDFS service for Parrot. As with other Parrot services, the

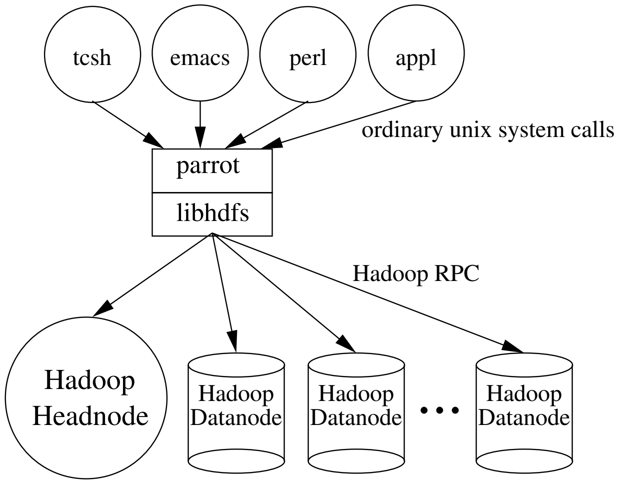


Figure 2: In the *parrot_hdfs* tool, we connect Parrot directly to Hadoop using *libhdfs*, which allows client applications to communicate with a HDFS service.

HDFS Parrot module (*parrot_hdfs*) allows unmodified end user applications to access remote filesystems, in this case HDFS, transparently and without any special priviledges. This is different from tools such as FUSE which require kernel modules and thus adminstrative access. Since Parrot is a static binary and requires no special priviledges, it can easily be deployed around our campus grid and used to access remote filesystems.

Figure 2 shows the architecture of the *parrot_hdfs* module. Applications run normally until an I/O operation is requested. When this occurs, Parrot traps the system call using the *ptrace* system interface and reroutes to the appropriate filesystem handler. In the case of Hadoop, any I/O operation to the Hadoop filesytem is implemented using the *libhdfs* library. Of course, the trapping and redirection of system calls incurs some overhead, but as shown in a previous paper [4], the *parrot_hdfs* module is quite serviceable and provides adequate performance.

Unfortunately, the *parrot_hdfs* module was not quite a perfect solution for integrating Hadoop into our campus infrastructure. First, while Parrot is lightweight and has minimal dependencies, *libhdfs*, the library required for communicating to Hadoop, does not. In fact, to use *libhdfs* we must have access to a whole Java virtual machine installation, which is not always possible in a heterogeneous campus grid with multiple administrative domains.

Second, only sequential writing was supported by the module. This is because Hadoop, by design, is a write-once-read-many filesystem and thus is optimized for streaming reads rather than frequent writes. Moreover, due to the instability of the append feature in the version of Hadoop deployed in our cluster, it was also not possible to support appends to existing files. Random and sequential reads, however, were fully supported. The *parrot_hdfs* module did not provide a means of overcoming this limitation.

Finally, the biggest impediment to using both Hadoop and the *parrot_hdfs* module was the lack of proper access controls. Hadoop's permissions can be entirely ignored as there is no authentication mechanism. The filesystem accepts whatever identity the user presents to Hadoop. So while a user may partition the Hadoop filesystem for different groups and users, any user may report themselves as any identity thus nullifying the permission model. Since *parrot_hdfs* simply provides the underlying access control system, this problem is not mitigated in anyway by the service. For a small isolated cluster filesystem this is not a problem, but for a large campus grid, such a lack of permissions is worrisome and a deterent for wide adoption, especially if sensitive research data is being stored.

So although *parrot_hdfs* is a useful tool for integrating Hadoop into a campus grid environment, it has some limitations that prevent it from being a complete solution. Moreover, Hadoop itself has limitations that need to be addressed before it can be widely deployed and used in a campus grid.

3.2 Parrot + Chirp + HDFS

To overcome these limitations in both *parrot_hdfs* and Hadoop, we decided to connect Parrot to Chirp and then Hadoop. In order to accomplish this, we had to implement a new virtual multiplexing interface in Chirp to allow it to to use any backend filesystem. With this new capability, a Chirp server could act as a frontend to potentially any other filesystem, such as Hadoop, which is not usually accessible with normal Unix I/O system calls.

In our modified version of Chirp, the new virtual multiplexing interface routes each I/O Remote Procedure Call (RPC) to the appropriate backend filesystem. For Hadoop, all data access and storage on the Chirp server is done through Hadoop's *libhdfs* C library. Figure 1b illustrates how Chirp operates with Hadoop as a backend filesystem.

With our new system, a simple command line switch changes the backend filesystem during initialization of the server. The command to start the Chirp server remains simple to start and deploy:

```
$ chirp_server -f hdfs
               -x headnode.hadoop:9100
```

Clients that connect to the Chirp server such as the one above will work over HDFS instead of on the the server's local filesystem. The head node of Hadoop and port it listens on are specified using the `-x <host>:<port>` switch. The root of the Hadoop filesystem is the exported directory by Chirp changed via the `-r` switch. Because Hadoop and Java require multiple inconvenient environment variables to be setup prior to execution, we have also provided the `chirp_server_hdfs` wrapper which sets these variables to configuration defined values before running `chirp_server`.

Recall that Hadoop has no internal authentication system; it relies on external mechanisms. A user may declare any identity and group membership when establishing a connection with the Hadoop cluster. The model naturally relies on a trustworthy environment; it is not expected to be resilient to malicious use. In a campus environment, this may not be the case. So, besides being an excellent tool for grid storage in its own right, Chirp brings the ability to wall off Hadoop from the campus grid while still exposing it in a secure way. To achieve this, we would expect the administrator to firewall off regular Hadoop access while still exposing Chirp servers running on top of Hadoop. This setup is a key difference to *parrot_hdfs*, discussed in the previous section. Chirp brings its strong authentication mechanisms and flexible access control lists to protect Hadoop from unintended or accidental access whereas *parrot_hdfs* connects to an exposed, unprotected Hadoop cluster.

3.3 Limitations when Bridging Hadoop with Chirp

The new modified Chirp provides users with an approximate POSIX interface to the Hadoop filesystem. Certain restrictions are unavoidable due to limitations stemming from design decisions in HDFS. The API provides most POSIX operations and semantics that are needed for our bridge with Chirp, but not all. We list certain notable problems we have encountered here.

- **Errno Translation** The Hadoop *libhdfs* C API binding interprets exceptions or errors within the Java runtime and sets errno to an appropriate value. These translations are not always accurate and some are missing. A defined *errno* value within *libhdfs* is *EINTERNAL* which is the generic error used when another is not found to be more appropriate. Chirp tries to sensibly handle these errors by returning a useful status to the user. Some errors must be caught early because Hadoop does not properly handle all error conditions. For example, attempting to open a file that is a directory results *errno* being set to *EINTERNAL* (an HDFS generic *EIO* error) rather than the correct *EISDIR*.

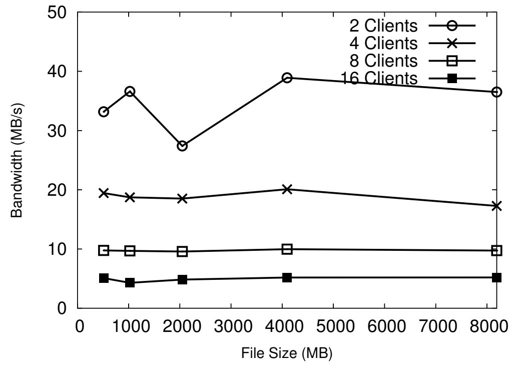
- **Appends to a File** Appends are largely unsupported in Hadoop within production environments. Hadoop went without appends for a long time in its history because the Map Reduce abstraction did not benefit from it. Still, use cases for appends surfaced and work to add support – and determine the appropriate semantics – has been in progress since 2007. The main barriers to its developement are coherency semantics with replication and correctly dealing with the (now) mutable last block of a file. As of now, turning appends on requires setting experimental parameters in Hadoop. Chirp provides some support for appends by implementing the operation through other primitives within Hadoop. That is, Chirp copies the file into memory, deletes the file, writes the contents of memory back to the file, and returns the file for further writes – effectively allowing appends. Support for appends to small files fit most use cases, logs in particular. Appending to a large file is not advised. The solution is naturally less than optimal but is necessary until Hadoop provides non-experimental support.

- **Random Writes** Random writes to a file are not allowed by Hadoop. This type of restriction generally affects programs, like linkers, that do not write to a file sequentially. In Chirp, writes to areas other than the end of the file are silently changed to appends instead.

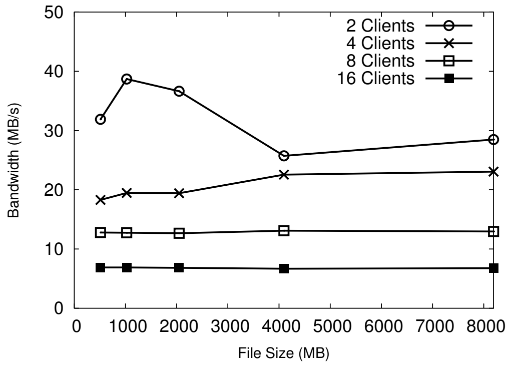
- **Other POSIX Incompatibilities** Hadoop has other peculiar incompatibilities with POSIX. The execute bit for regular files is nonexistent in Hadoop. Chirp gets around this by reporting the execute bit set for all users. Additionally, it does not allow renaming a file to one that already exists. This would be equivalent to the Unix *mv* operation from one existing file to another existing file. Further, opening, unlinking, and closing a file will result in Hadoop failing with an *EINTERNAL* error. For these types of semantic errors, the Chirp server and client have no way of reasonably responding or anticipating such an error. We expect that as the HDFS binding matures many of these errors will be responded to in a more reasonable way.

4 Evaluation

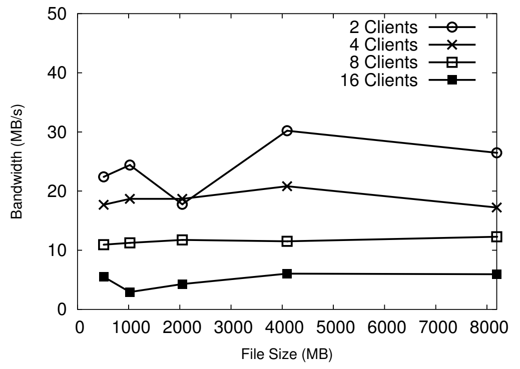
To evaluate the performance characteristics of our Parrot + Chirp + HDFS system, we ran a series of benchmark tests. The first is a set of micro-benchmarks designed to test the overhead for individual I/O system calls when using Parrot with various filesystems. The second is a data transfer performance test of our new system versus the native Hadoop tools. Our tests ran on a 32 node cluster of machines (located off Notre Dame's main campus) with the Hadoop setup using 15 TB of aggregate capacity. The cluster is using approximately 50% of available space. The Hadoop cluster shares a 1 gigabit link to communicate with the main campus network. We have setup a Chirp server as a frontend to the Hadoop cluster on one of the data nodes.



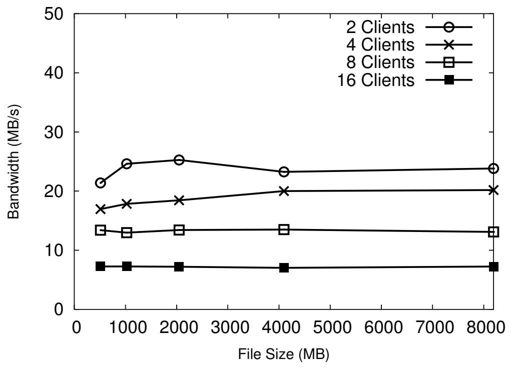
(a) Campus Network Reads to Chirp+HDFS.



(b) Campus Network Reads to HDFS.



(c) Campus Network Writes to Chirp+HDFS.



(d) Campus Network Writes to HDFS.

Figure 3

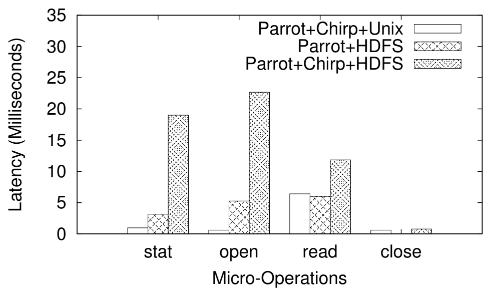


Figure 5: Latency of I/O Operations using Microbench

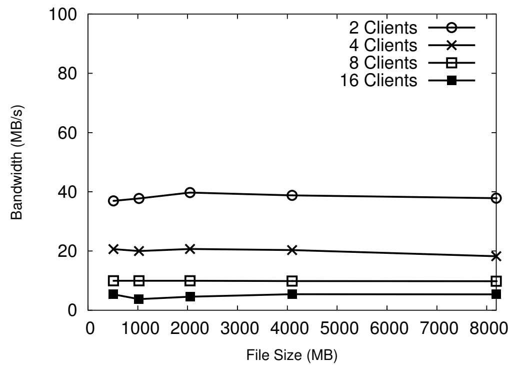
The results of these micro-benchmarks are shown in Figure 5. As can be seen, our new Parrot + Chirp + HDFS systems has some overhead when compared to accessing a Chirp server or HDFS service using Parrot. In particular, the metadata operations of *stat* and *open* are particularly heavy in our new system while *read* and *close* are relatively close. The reason for this high overhead on metadata-intensive operations is because of the access control lists Chirp uses to implement security and permissions. For each *stat* or *open*, multiple files must be opened and checked in order to verify valid access, which increases the latencies for these system calls.

4.1 Micro-benchmarks

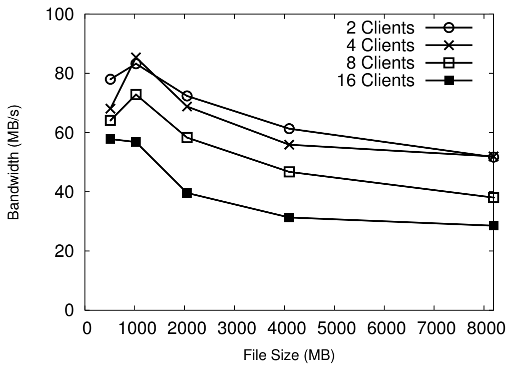
The first set of tests we ran were a set of micro-benchmarks that measure the average latency of various I/O system calls such as *stat*, *open*, *read*, *close*. For this test we wanted to know the cost of putting Chirp in front of HDFS and using Parrot to access HDFS through our Chirp server. We compared these latencies to those for accessing a Chirp server hosting a normal Unix filesystem through Parrot, and accessing HDFS directly using Parrot.

4.2 Performance benchmarks

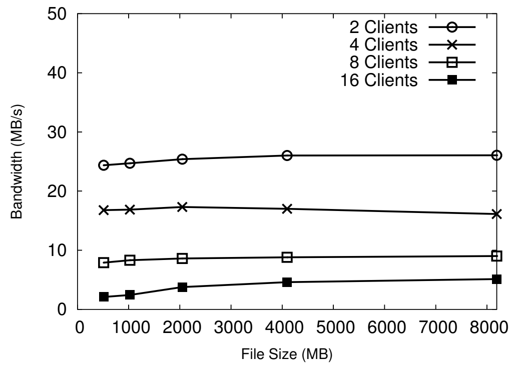
For our second set of benchmarks, we stress test the Chirp server by determining the throughput achieved when putting and getting very large files. These types of commands are built into Chirp as the RPC's `getfile` and `putfile`. Our tests compared the equivalent Hadoop native client `hadoop fs get` and `hadoop fs put` commands to the Chirp ones. The computer on the campus network initiating the tests used the Chirp command



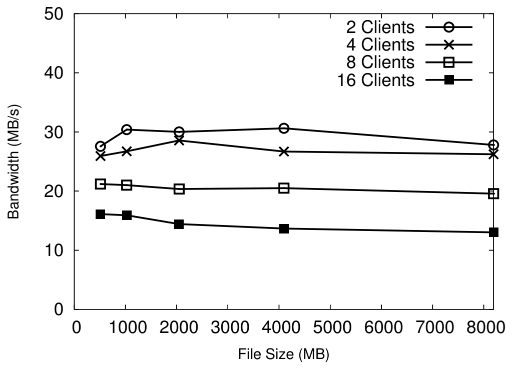
(a) LAN Network Reads to Chirp+HDFS.



(b) LAN Network Reads to HDFS.



(c) LAN Network Writes to Chirp+HDFS.



(d) LAN Network Writes to HDFS.

Figure 4

line tool, which uses the *libchirp* library without any external dependencies, and the Hadoop native client, which requires all the Hadoop and Java dependencies. We want to test how the Chirp server performs under load compared to the Hadoop cluster by itself. For these tests, we communicated with the Hadoop cluster and the Chirp server frontend from the main campus. All communication went through the 1 gigabit link to the cluster.

The graphs in Figures 3a and 3b show the results of reading (getting) a file from the Chirp server frontend and via the the native Hadoop client, respectively. We see that the Chirp server is able to maintain throughput comparable to the Hadoop native client; the Chirp server does not present a real bottleneck to reading a file from Hadoop. In Figures 3c and 3d, we have the results of writing (putting) a file to the Chirp server frontend and native Hadoop client, again respectively. These graphs similarly show that the Chirp server obtains similar throughput to the Hadoop native client for writes. For these tests, the 1 gigabit link plays a major role in limiting the exploitable parallelism obtainable via writing to Hadoop over a faster (local) network. Despite hamstringing Hadoop's potential parallelism, we find these tests valuable because the campus network machines will follow access the Hadoop storage in this way. For contrast, in the next tests, we remove this barrier.

Next, we place the clients on the same LAN as the Hadoop cluster and the Chirp server frontend. We expect that throughput will not be hindered by the link rate of the network and parallelism will allow Hadoop to scale. Similar to the tests from the main campus grid, the clients will put and get files to the Chirp server and via the Hadoop native client.

The graphs in Figures 4a and 4b show the results of reading a file over the LAN setup. We see that we get similar throughput over the Chirp server as in the main campus tests. While the link to the main campus grid is no longer a limiting factor, the Chirp server's machine still only has a gigabit link to the LAN. Meanwhile, Hadoop achieves respectably good performance, as expected, when reading files with a large number of clients. The parallelism gives Hadoop almost twice the throughput. For a larger number of clients, Hadoop's performance outstrips chirp by a large factor of 10. The write performance in Figures 4c and 4d, show similar results. The writes over the LAN bottleneck on the gigabit link to the Chirp server while Hadoop maintains minor performance hits as the number of clients increase. In both cases, the size of the file did not have a significant effect on the average throughput. Because one (Chirp) server cannot handle the throughput due to network link restrictions in a LAN environment, we recommend adding more Chirp servers on other machines in the LAN to help distribute the load.

We conclude from these benchmark evaluations that Chirp achieves adequate performance compared to Hadoop with some overhead. In a campus grid setting we see that Chirp gets approximately equivalent streaming throughput compared to the native Hadoop client. We find this acceptable for the type of environment we are targeting.

5 Conclusions

Hadoop is a highly scalable filesystem used in a wide array of disciplines but lacks support for campus or grid computing needs. Hadoop lacks strong authentication mechanisms and access control. Hadoop also lacks ease of access due to the number of dependencies and program requirements. We have designed a solution that enchances the functionality of Chirp, coupled with Parrot or FUSE, to provide a bridge between a user's work and Hadoop. This new system can be used to correct these problems in Hadoop so it may be used safely and securely on the grid without significant loss in performance.

The new version of the Chirp software is now capable of multiplexing the I/O operations on the server so operations are handled by filesystems other than the local Unix system. Chirp being capable of bridging Hadoop to the grid for safe and secure use is a particularly significant consequence of this work. Chirp can be freely downloaded at: http://www.cse.nd.edu/ ccl/software.

References

[1] Filesystem in user space. http://sourceforge.net/projects/fuse.

[2] Boilergrid web site. http://www.rcac.purdue.edu/userinfo/resources/boilergrid, 2010.

[3] D. Borthakur. The hadoop distributed file system: Architecture and design. http://hadoop.apache.org/, 2007.

[4] H. Bui, P. Bui, P. Flynn, and D. Thain. ROARS: A Scalable Repository for Data Intensive Scientific Computing. In *The Third International Workshop on Data Intensive Distributed Computing at ACM HPDC 2010*, 2010.

[5] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large cluster. In *Operating Systems Design and Implementation*, 2004.

[6] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A Security Architecture for Computational Grids. In *ACM Conference on Computer and Communications Security*, pages 83–92, San Francisco, CA, November 1998.

[7] S. Ghemawat, H. Gobioff, and S. Leung. The Google filesystem. In *ACM Symposium on Operating Systems Principles*, 2003.

[8] Hadoop. http://hadoop.apache.org/, 2007.

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

[10] M. C. Schatz. CloudBurst: highly sensitive read mapping with MapReduce. *Bioinformatics*, 25(11):1363–1369, 2009.

[11] J. Steiner, C. Neuman, and J. I. Schiller. Kerberos: An authentication service for open network systems. In *Proceedings of the USENIX Winter Technical Conference*, pages 191–200, 1988.

[12] D. Thain and M. Livny. Parrot: Transparent user-level middleware for data-intensive computing. Technical Report 1493, University of Wisconsin, Computer Sciences Department, December 2003.

[13] D. Thain and M. Livny. How to Measure a Large Open Source Distributed System. *Concurrency and Computation: Practice and Experience*, 18(15):1989–2019, 2006.

[14] D. Thain and C. Moretti. Efficient access to many small files in a filesystem for grid computing. In *IEEE Conference on Grid Computing*, Austin, TX, September 2007.

[15] D. Thain, C. Moretti, and J. Hemmes. Chirp: A practical global file system for cluster and grid computing. *Journal of Grid Computing*, to appear in 2008.

[16] D. Thain, T. Tannenbaum, and M. Livny. Condor and the grid. In F. Berman, G. Fox, and T. Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley, 2003.