HFAA: A Generic Socket API for Hadoop File Systems

Adam Yee
University of the Pacific
Stockton, CA
adamjyee@gmail.com

Jeffrey Shafer
University of the Pacific
Stockton, CA
jshafer@pacific.edu

ABSTRACT

Hadoop is an open-source implementation of the MapReduce programming model for distributed computing. Hadoop natively integrates with the Hadoop Distributed File System (HDFS), a user-level file system. In this paper, we introduce the Hadoop Filesystem Agnostic API (HFAA) to allow Hadoop to integrate with any distributed file system over TCP sockets. With this API, HDFS can be replaced by distributed file systems such as PVFS, Ceph, Lustre, or others, thereby allowing direct comparisons in terms of performance and scalability. Unlike previous attempts at augmenting Hadoop with new file systems, the socket API presented here eliminates the need to customize Hadoop's Java implementation, and instead moves the implementation responsibilities to the file system itself. Thus, developers wishing to integrate their new file system with Hadoop are not responsible for understanding details of Hadoop's internal operation.

In this paper, an initial implementation of HFAA is used to replace HDFS with PVFS, a file system popular in high-performance computing environments. Compared with an alternate method of integrating with PVFS (a POSIX kernel interface), HFAA increases write and read throughput by 23% and 7%, respectively.

1. INTRODUCTION

Hadoop is an open-source framework that implements the MapReduce parallel programming model [3, 10]. The Hadoop framework is composed of a MapReduce engine and a user-level file system. For portability and ease of installation, both components are written in Java and only require commodity hardware. In recent years, Hadoop has become popular in industry and academic circles [8]. For example, as of late 2010, Yahoo had over 43,000 nodes running Hadoop for both research and production application [14].

The Hadoop Distributed File System (HDFS) manages storage resources across a Hadoop cluster by providing global access to any file [4, 16]. HDFS is implemented by two services: one central *NameNode* and many *DataNodes*. The NameNode is responsible for maintaining the HDFS directory tree. Clients contact the NameNode in order to perform common file system operations, such as open, close, rename, and delete. The NameNode does not store HDFS data itself, but rather maintains a mapping between HDFS file name, a list of blocks in the file, and the DataNode(s) on which those blocks are stored.

Although HDFS stores file data in a distributed fashion, file metadata is stored in the centralized NameNode service. While sufficient for small-scale clusters, this design prevents Hadoop from scaling beyond the resources of a single NameNode. Prior analysis of CPU and memory requirements for the NameNode revealed that this service is memory limited. A large NameNode with 60GB of RAM could store at most 100 million files averaging 128MB (2 HDFS blocks) in size. Further, with a 30% target CPU load for low-latency service, such a NameNode could support a cluster with 100,000 readers but only 10,000 writers [17].

Beyond improved scalability (a goal also shared by the HDFS "Federation" approach of using several independent namespaces stored on separate NameNodes [18]), there are other motivations for replacing HDFS. For example, alternate file systems allow for write and rewrite anywhere operations (increasing application flexibility) and support remote DMA transfers across networks like InfiniBand (increasing data transfer performance). Beyond scalability, performance, and feature-set support, one key driver for the use of alternate file systems is simply that these file systems are already widely deployed. In many high-performance computing environments, Hadoop (i.e. MapReduce) is simply the latest in a wide variety of application programming styles to be supported, placing an emphasis on compatibility with existing infrastructure.

Recent work has investigated replacing HDFS with other distributed file systems. To date, Hadoop has been integrated with Amazon S3 [2], CloudStore (aka KosmosFS) [5], Ceph [12], GPFS [9], Lustre [1], and PVFS [19]. Each of these implementations required special-purpose code for integration with Hadoop, and required the developer to be familiar with the internal operations of both Hadoop and the target filesystem.

This paper evaluates three different methods to replace HDFS with other distributed file systems. The first method uses a *POSIX driver* to directly mount a distributed file system on each cluster node. This driver is typically provided with the file system, and is intended to allow easy integration with any unmodified program or utility. When used by Hadoop, however, storage performance suffers due to limitations of the POSIX interface. For example, Hadoop is unable to query the file system for file locality information, and thus cannot schedule computations to minimize network data transfer.

The second method, *shim code*, extends the Hadoop Java implementation (by modifying the convenient `FileSystem` abstract class that Hadoop provides) to directly access another file system in user space. Such shim code is file system specific, *i.e.,* the PVFS shim only works with PVFS.

Due to these limitations, a third method – the *Hadoop Filesystem Agnostic API (HFAA)* – is developed in order to overcome scalability limitations of HDFS and allow the direct integration of alternate file systems. HFAA provides a universal, generic interface that allows Hadoop to run on any file system that supports network sockets, particularly file systems without a single point of failure and more general purpose file system semantics. Its design moves integration responsibilities outside of Hadoop, and does not require user or developer knowledge of the Hadoop MapReduce framework. Essentially, Hadoop integration is reduced to one API.

The remainder of this paper is organized as follows. First, Section 2 describe Hadoop and HDFS, its distributed file system. Next, Section 3 contrasts HDFS against other distributed file systems. Then, Section 4 describes the design of the proposed HFAA architecture. Section 5 benchmarks an implementation of HFAA with OrangeFS (a PVFS fork) and compares its performance against other approaches. Finally, Section 6 discusses related work to our approach, and Section 7 concludes this paper.

2. HADOOP ARCHITECTURE

The Hadoop framework is implemented as two key services: the Hadoop MapReduce engine and the Hadoop Distributed File System (HDFS). Although they are typically used together, each can be operated independently if desired. For example, users of Amazon's Elastic MapReduce service use the Hadoop MapReduce engine in conjunction with Amazon's own Simple Storage Service (S3) [2].

In Hadoop, the MapReduce engine is implemented by two software services, the *JobTracker* and *TaskTracker*. The centralized JobTracker runs on a dedicated cluster node and is responsible for splitting the input data into pieces for processing by independent map and reduce tasks (by coordinating with the user-level file system), scheduling each task on a cluster node for execution, monitoring execution progress by receiving heartbeat signals from cluster nodes, and recovering from failures by re-running tasks. On each cluster node, an instance of the TaskTracker service accepts map and reduce tasks from the JobTracker, executes the tasks, and reports the status back to the JobTracker.

HDFS provides global access to any file in a shared namespace across the Hadoop cluster [4, 16]. HDFS is implemented by two services: the *NameNode* and *DataNode*.

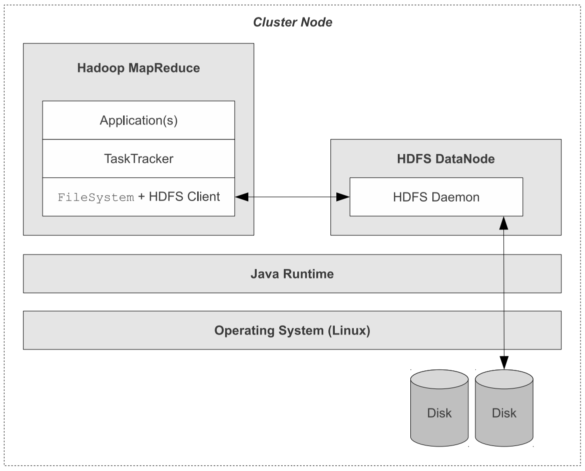

Figure 1: Hadoop Cluster Node (non-master)

The *NameNode* is responsible for maintaining the HDFS directory tree, and is a centralized service in the cluster operating on a single node. Clients contact the NameNode in order to perform common file system operations, such as open, close, rename, and delete. The NameNode does not store HDFS data itself, but rather maintains a mapping between HDFS file name, a list of blocks in the file, and the DataNode(s) on which those blocks are stored.

In addition to a centralized NameNode, all remaining cluster nodes provide the *DataNode* service. Each DataNode stores HDFS blocks (64MB chunks of a single logical file) on behalf of local or remote clients. Each block is saved as a separate file in the node's local file system, which uses a native file system like ext4. Blocks are created or destroyed on DataNodes at the request of the NameNode, which validates and processes requests from clients. Although the NameNode manages the namespace, clients communicate directly with DataNodes to read or write data at the HDFS block level.

Figure 1 shows the architecture of a standard Hadoop cluster node used for both computation and storage. The MapReduce engine (running inside a Java virtual machine) executes the user application. When the application reads or writes data, requests are passed through the Hadoop `org.apache.hadoop.fs.FileSystem` class, which provides a standard interface for distributed file systems, including the default HDFS. An HDFS client is then responsible for retrieving data from the distributed file system by contacting a DataNode with the desired block. In the common case, the DataNode is running on the same node, so no external network traffic is necessary. The DataNode, also running inside a Java virtual machine, accesses the data stored on local disk using normal file I/O functions.

3. SCALABLE FILE SYSTEMS

There are many distributed file systems designed to scale across a large cluster computer. Some are open source, including Ceph [20], Lustre [6], and PVFS [7], while others are proprietary, including GPFS [15], the Google filesystem [11, 13], and PanFS [21].

Function	HDFS	Lustre	Ceph	PVFS
Namespace Service	Centralized	Centralized	Distributed	Distributed
Fault Tolerance	File replication and auto-recovery	Hardware dependant (RAID)	File replication and auto-recovery	Hardware dependant (RAID)
Write Semantics	Write-once (file append provisionally supported)	Write and re-write anywhere with most POSIX semantics		
Accessibility	Custom Hadoop API or kernel driver (non-POSIX)	Custom (Lustre/Ceph/PVFS) API or kernel driver providing POSIX-like functionality		
Deployment Model	Nodes used for computation and storage	Separate nodes for computation or storage (dedicated services)		

Table 1: Comparison of HDFS and other Distributed File Systems

Table 1 provides a comparison between HDFS and other open-source distributed file systems including Ceph, Lustre, and PVFS. Key differences include the handling of file metadata in a centralized or distributed fashion, the implementation of fault tolerance by either hardware RAID (below the level of the file system) or software-based replication, and the imposition of non-POSIX restrictions such as write-once (and only once) files. Broadly, HDFS was designed specifically to support one kind of programming model – MapReduce – and thus features a simplified design optimized heavily towards streaming accesses. In contrast, other parallel distributed file systems were designed for high-performance computing environments where multiple programming models – such as MPI, PVM, and OpenMP – are commonplace, and thus must support a broader range of features.

4. HFAA API

The Hadoop Filesystem Agnostic API (HFAA) provides a simple way for Hadoop to interface with file systems other than HDFS. It is targeted at developers of these alternate file systems, *i.e.,* those who have an understanding of their distributed file system, but are not experts in the internal operation of Hadoop. HFAA eliminates the need for custom extensions to Hadoop's Java implementation, and allows communication with any file system over TCP sockets. The overall architecture of HFAA is described here. For the purpose of design discussion, distributed and parallel file systems other than the native Hadoop Distributed File System (HDFS) (such as PVFS, Ceph, or Lustre) will be referred to by the generic name *NewDFS*.

This API is designed for deployment in the typical Hadoop environment where each cluster node is responsible for both computation (via Hadoop MapReduce) and storage (via HDFS). When HDFS is replaced with whatever "NewDFS" is desired, each node will still be responsible for both computation and storage. The HDFS daemon will be disabled, and a NewDFS daemon configured in its place.

To enable Hadoop MapReduce to communicate with the NewDFS daemon, the HFAA architecture adds two software components to the system, an *HFAA client* and an *HFAA server*, as shown in Figure 2. The HFAA client integrates with the standard Hadoop MapReduce framework and intercepts read/write requests that would ordinarily go to HDFS. The client then forwards the requests via network

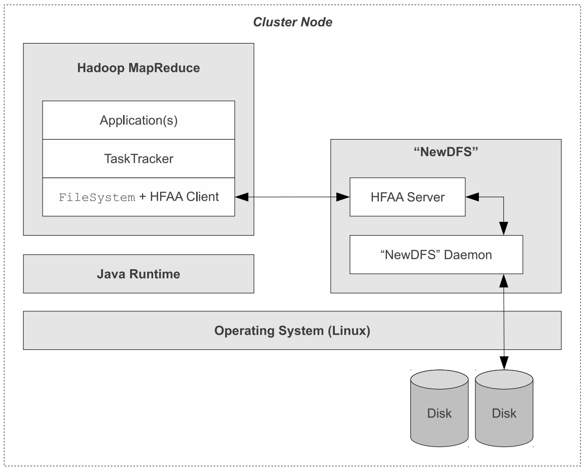

Figure 2: HFAA Cluster Node (non-master)

sockets to the HFAA server, which typically will be running on the same node, but could be remotely located. The HFAA server is responsible for interfacing with NewDFS. It receives network requests from the HFAA client to perform file system operations (*e.g.,* create, open, delete, etc.) and responds with the data/metadata results from NewDFS.

The HFAA client is written in Java and integrated into the Hadoop framework. The client can be re-used with any NewDFS. In contrast, the HFAA server is written in whatever language NewDFS is implemented in, and must be re-written for each new file system that is integrated. We envision this component being provided by developers of NewDFS, who have no need for detailed knowledge of Hadoop's inner workings. Details of the design and operation of the HFAA client and server are described next.

4.1 Client: Integration with Hadoop

The Hadoop implementation in Java includes an abstract class called `org.apache.hadoop.fs.FileSystem`. Several file systems extend and implement this abstract class, including HDFS (the default file system), Amazon S3, and the Kosmos File System. The HFAA client integrates with Hadoop by implementing this `FileSystem` class, essentially replacing the HDFS interface with equivalent functionality to interface with any NewDFS over TCP sockets.

Function	Description
open	Open a file to read.
create	Create a file to write.
append	Append data to an existing file.
rename	Change a file name and/or move the file to a different directory.
delete	Delete a file/directory and its contents.
listStatus	List the files within a directory.
mkdirs	Make a directory for a given file path. Non existent parent directories must also be created.
getFileStatus	Fetch metadata (size, block location, replication, owner, ...) for a given file or directory.
write	Write functionality is implemented within a new `OutputStream` class.
read	Read functionality is implemented within a new `InputStream` class.

Table 2: HFAA Operations

The key functionality that HFAA implements includes is shown in Table 2. To make the protocol generic, simple command request and response type codes and their data/metadata are exchanged between the HFAA client and HFAA server in string format. Parsing requests and responses is flexible and easy to work with since most programming languages offer libraries for string manipulation. For full specifications on the HFAA protocol, refer to [22].

4.2 Server: Integration with NewDFS

The HFAA server interfaces with NewDFS. It can receive network requests from the HFAA client to perform file system operations (*e.g.,* create, open, delete, etc.) and responds with the proper data/metadata obtained from NewDFS. In this design, the interface between the HFAA client and HFAA server are standardized, but the interface between the HFAA server and the NewDFS daemon are left up to the implementer. Thus, the HFAA server could be integrated into the NewDFS daemon itself, or left as a stand-alone program.

The basic structure of the HFAA server, regardless of whether it is standalone or integrated with the NewDFS daemon, is as follows. The HFAA server listens for incoming TCP requests in a tight loop and passes each request to an independent thread or process for completion. The HFAA server must be able to respond to multiple requests concurrently because a typical Hadoop node is processing several MapReduce tasks in parallel, and because each task has at least one data file and several metadata files (tracking task completion status) open at any given time. In each process are a set of request handlers for each Hadoop API function (*e.g.,* getFileStatus(), listStatus()). The request handlers service the request until completion, and then exit.

5. EVALUATION

The HFAA client and server architecture described in Section 4 was implemented and tested with Hadoop 0.20.204.0 and OrangeFS 2.8.4, a distributed file system that is a branch of PVFS. The HFAA client was written in Java, because it is compiled into the Hadoop framework. In contrast, the HFAA server was written in C to take advantage of existing PVFS code. For implementation convenience, the HFAA server was not directly integrated into the PVFS storage daemon. Rather, it was implemented as a standalone process similar in style to the existing `admintools` provided with the new file system. These utilities offer POSIX-style commands (*e.g.,* `ls`, `touch`) for PVFS maintenance and administration.

Four homogeneous servers were used for benchmarking. Each server consists of an Intel Xeon X3430 processor, 4GB of RAM, and a 500GB SATA hard drive. A 100GB partition on the outer edge of the disk (*i.e.,* the fastest region) was allocated to store HDFS or PVFS data. Using only a small region of the disk allows the access bandwidth difference between the fastest and slowest sectors due to physical disk location to be minimized in these experiments to under 8%. Each server was configured with the Ubuntu 10.10 Linux distribution and a software stack (Hadoop + distributed file system) that varies by experiment. All nodes were connected by a gigabit Ethernet switch on a private network.

Table 3 describes the three different cluster software configurations used for experiments: Hadoop with HDFS, Hadoop with PVFS, and Hadoop with PVFS using HFAA. For consistency across experiments, in all configurations a single "master" server was used for job scheduling, and three "slave" servers were used for computation and to store file data. The location of file metadata varied between experiments.

The first configuration, labeled *Hadoop+HDFS*, is the standard Hadoop architecture, where the master node runs the centralized JobTracker and NameNode services (for job management and file system namespace), and each slave node runs its own instance of the TaskTracker and DataNode services (for computation and distributed storage).

In the second configuration, labeled *Hadoop+PVFS*, HDFS has been removed and replaced with PVFS. A POSIX client driver, provided with PVFS, is used to mount the distributed file system on each node, and Hadoop is configured to use this "local" file system directly via its `RawLocalFileSystem` interface. In this configuration, Hadoop is limited by the POSIX driver interface, and is unable to obtain locality information from the underlying file system.

The PVFS architecture is designed with 3 components: metadata servers (storing metadata), I/O servers (storing data), and clients. For consistency with the HDFS configuration, the master node in the Hadoop+PVFS configuration is used only to run the metadata server, while all other slave nodes runs both a metadata server and an I/O server. Thus, only the 3 slaves store file data, as in the HDFS configuration. Unlike HDFS, however, the Metadata Server(s) (the NameNode equivalent) is *distributed* over the entire cluster.

Finally, in the third *Hadoop+HFAA+PVFS* configuration, the interface between Hadoop and PVFS is changed. The POSIX driver and PVFS *client* is removed, and replaced by the new HFAA client and server architecture discussed in Section 4. As in the previous configuration, each node runs a Metadata Server, but only the slave nodes run an I/O server, in order to emulate the original HDFS architecture.

Node	Hadoop+HDFS	Hadoop+PVFS	Hadoop+HFAA+PVFS
Master (1)	H: JobTracker D: NameNode **(metadata)**	H: JobTracker P: POSIX Driver P: Client P: Metadata Server **(metadata)**	H: JobTracker A: HFAA Client A: HFAA Server P: Metadata Server **(metadata)**
Slaves (3)	H: TaskTracker D: DataNode **(data)**	H: TaskTracker P: POSIX Driver P: Client P: Metadata Server **(metadata)** P: I/O Server **(data)**	H: TaskTracker A: HFAA Client A: HFAA Server P: Metadata Server **(metadata)** P: I/O Server **(data)**

Table 3: Cluster Software Configuration (H=Hadoop MapReduce, D=HDFS, P=PVFS, A=HFAA)

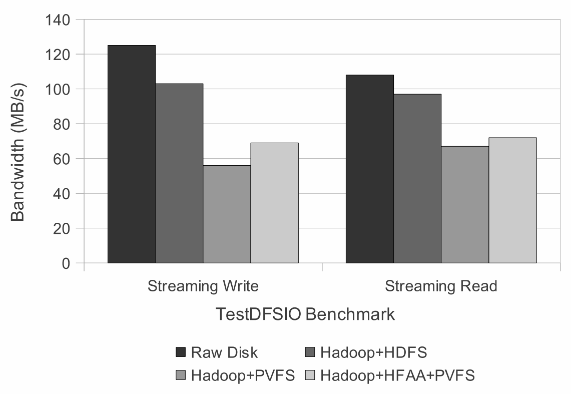

Figure 3: File Access Bandwidth (Read and Write)

Each of the three cluster configurations are benchmarked with a MapReduce application called `TestDFSIO`, which is provided with the Hadoop framework. This benchmark performs parallel streaming read or streaming write operations to many files across the cluster. Each individual file is read (or written) by a single TestDFSIO map task, and each cluster slave can run several map tasks concurrently. As each map task completes across the cluster, a single reduce task aggregates and reports performance statistics. In these experiments, the TaskTracker on each slave node runs a single map task with TestDFSIO either reading or writing a single 5GB file in the global file system, for a total of 15GB of data accessed in aggregate across the cluster. Note that because replication is not natively supported in PVFS, replication was disabled in all configurations.

Figure 3 shows the TestDFSIO results for the three cluster configurations for both reading and writing workloads. These results are compared to the performance of the raw disk outside of the Hadoop environment entirely, which was measured via the `dd` utility configured to do streaming accesses. The standard Hadoop+HDFS configuration achieved 82% of the average write throughput and 87% of the average read throughput of the raw disk. In contrast, the Hadoop+PVFS configuration only achieved 46% of the average write throughput and 60% of the average read throughput, compared to the raw disk bandwidth. Finally the Hadoop+HFAA+PVFS configuration achieved 55% of the average write throughput and 64% of the average read throughput compared to the original raw disk bandwidth, showing how HFAA can preserve the original file system bandwidth and make it accessible to Hadoop.

The results demonstrate the functionality of the HFAA interface. Although using this API with PVFS did not match or surpass the I/O rates of HDFS, 67% of HDFS write throughput and 74% of HDFS read throughput is maintained. Integrating PVFS with HFAA achieves a 23% gain on write throughput and a 7% gain on read throughput compared to using PVFS as a local file system through the POSIX kernel interface.

6. RELATED WORK

Integrating the MapReduce functionality of Hadoop with file systems other than HDFS has been of significant interest in recent years. This integration typically follows one of two possible approaches: extending Hadoop's built-in `FileSystem` abstract class with shim code to support the desired distributed file system, or using a POSIX driver to expose the distributed file system to the underlying OS, and then directly accessing this from inside Hadoop.

Examples of alternate distributed file systems that have been successfully used with Hadoop MapReduce include Amazon S3 [2], Ceph [12], CloudStore/KosmosFS [5], GPFS [9], PVFS [19], and Lustre [1]. Many of these file systems can be used with either the POSIX driver approach or the shim code approach for higher performance. The key difference between all of these approaches and the approach chosen for HFAA is that HFAA was designed to keep the Hadoop-facing implementation constant, and push a simplified set of implementation responsibilities to the distributed file system. In contrast, all of the prior integration projects each involve their own separate, unique set of modifications and patches to the Hadoop code base.

7. CONCLUSIONS

The Hadoop Filesystem Agnostic API extends Hadoop with a simple communication protocol, allowing it to integrate with any file system which supports TCP sockets. This new API eliminates the need for customizing Hadoop's Java implementation, and instead moves future implementation responsibilities to the file system itself. Thus, developers wishing to integrate their new file system with Hadoop are not responsible for understanding details of Hadoop's internal operation.

By providing a file system agnostic API, future work can directly explore the performance tradeoffs of using the Map-

Reduce programming model with file systems supporting write-anywhere operations, distributed metadata servers, and other advances. Further, HFAA allows Hadoop to integrate more easily into existing high-performance computing environments, where alternate distributed file systems are already present.

For details on the HFAA design and protocol, refer to [22].

8. REFERENCES

[1] Using Lustre with Apache Hadoop. `http://wiki.lustre.org/images/1/1b/Hadoop_wp_v0.4.2.pdf`, Jan. 2010.

[2] Amazon S3. `http://wiki.apache.org/hadoop/AmazonS3`, 2012.

[3] Hadoop. `http://hadoop.apache.org`, 2012.

[4] HDFS (Hadoop distributed file system) architecture. `http://hadoop.apache.org/common/docs/current/hdfs_design.html`, 2012.

[5] kosmosfs - kosmos distributed filesystem. `http://code.google.com/p/kosmosfs/`, 2012.

[6] Lustre file system. `http://www.lustre.org`, 2012.

[7] Parallel virtual file system, version 2. `http://www.pvfs.org`, 2012.

[8] Powered by Hadoop. `http://wiki.apache.org/hadoop/PoweredBy`, 2012.

[9] R. Ananthanarayanan, K. Gupta, P. Pandey, H. Pucha, P. Sarkar, M. Shah, and R. Tewari. Cloud analytics: do we really need to reinvent the storage stack? In *Proceedings of the 2009 conference on Hot topics in cloud computing*, HotCloud'09, Berkeley, CA, USA, 2009.

[10] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th Symposium on Opearting Systems Design & Implementation*, pages 10–10, Berkeley, CA, USA, 2004.

[11] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, New York, NY, USA, 2003.

[12] C. Maltzahn, E. Molina-Estolano, A. Khurana, N. A. J., S. A. Brandt, and S. A. Weil. Ceph as a scalable alternative to the Hadoop distributed file system. In *login: The Magazine of USENIX*, volume 35, pages 38–49, Aug. 2010.

[13] M. K. McKusick and S. Quinlan. GFS: Evolution on fast-forward. *Queue*, 7:10:10–10:20, August 2009.

[14] S. Radia. HDFS federation - apache Hadoop india summit. `http://www.slideshare.net/huguk/hdfs-federation-hadoop-summit2011`, 2011.

[15] F. Schmuck and R. Haskin. Gpfs: A shared-disk file system for large computing clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST '02, Berkeley, CA, USA, 2002.

[16] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010.

[17] K. V. Shvachko. HDFS scalability: The limits to growth. In *login: The Magazine of USENIX*, volume 35, pages 6–16, Apr. 2010.

[18] K. V. Shvachko. Apache Hadoop: The scalability update. In *login: The Magazine of USENIX*, volume 36, pages 7–13, June 2011.

[19] W. Tantisiriroj, S. W. Son, S. Patil, S. J. Lang, G. Gibson, and R. B. Ross. On the duality of data-intensive file system design: reconciling HDFS and PVFS. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 67:1–67:12, New York, NY, USA, 2011.

[20] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: a scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 307–320, Berkeley, CA, USA, 2006.

[21] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou. Scalable performance of the panasas parallel file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 2:1–2:17, Berkeley, CA, USA, 2008.

[22] A. J. Yee. Sharing the love: A generic socket API for Hadoop mapreduce. Master's thesis, University of the Pacific, 2011.