The Hydra Filesystem: A Distributed Storage Framework

Benjamín González and George K. Thiruvathukal

Loyola University Chicago, Chicago IL 60611, USA,
{bgonzalez,gkt}@cs.luc.edu,
WWW home page: http://etl.cs.luc.edu

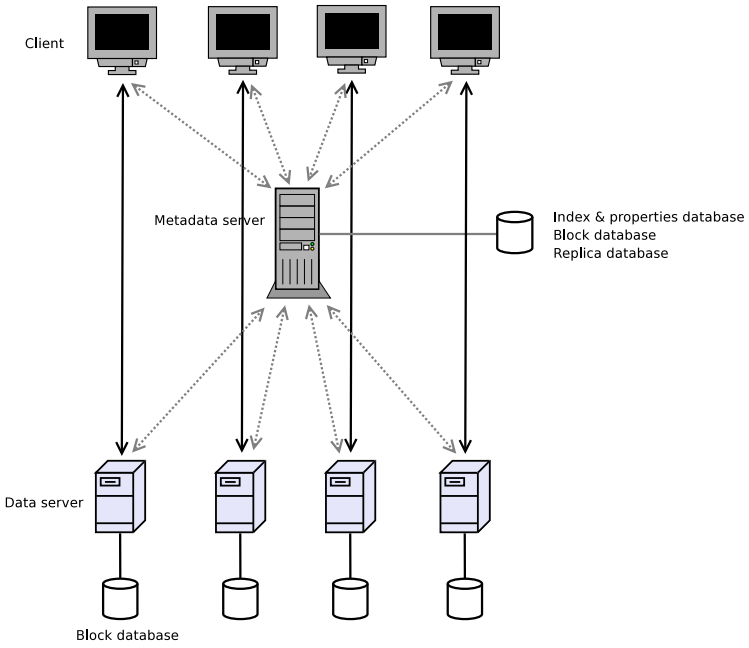
Abstract. Hydra File System (HFS) is an experimental framework for constructing parallel and distributed filesystems. While parallel and distributed applications requiring scalable and flexible access to storage and retrieval are becoming more commonplace, parallel and distributed filesystems remain difficult to deploy easily and configure for different needs. HFS aims to be different by being true to the tradition of high-performance computing while employing modern design patterns to allow various policies to be configured on a per instance basis (e.g. storage, communication, security, and indexing schemes). We describe a working prototype (available for public download) that has been implemented in the Python programming language.

1 Introduction

A distributed storage system supports data storage and retrieval among several computers or storage devices. The system is responsible for tracking object locations and synchronizing access between clients and the different nodes within a storage cluster. It should ensure consistency and availability at all times, even in the presence of individual node failures. HFS addresses these requirements and adds support for replicating data across the network to increase reliability and response time. HFS also has the ability to split data into a number of blocks of a specified size, thereby allowing several clients to perform parallel I/O to different regions of the same object.

While current research in parallel/distributed filesystems has focused on emulating the behavior of a file system over a set of distributed data nodes, our focus in this research is to provide a lower-level framework which is not tied to any particular indexing strategy whatsoever. Although our current testing suite emulates a filesystem, different ways of tracking and indexing objects can be implemented to fit other application requirements. For example, an overarching goal of this project is to make it possible with a common code base to build a flexible parallel filesystem while simultaneously making it possible to build a distributed storage system for simpler needs such as data backup and disaster recovery applications.

Fig. 1. Global architecture



2 Architectural Overview

HFS comprises three main components: the metadata server enforces system policies and orchestrates access between storage nodes and clients; the data server, whose current sole job is to provide access to the data itself and notify the metadata server when a block has been updated; and the client application programming interface (API), which requests data on behalf of the application and interacts with both the metadata and data server to fulfill that request.

HFS implements a communication layer over TCP/IP which allows any machine running the data server daemon to announce itself as a storage device to the metadata server, and upon success start serving both clients and other storage nodes.

The metadata server is able to index objects based on a given set of characteristics (e.g. filename and parent) and map them to object id's stored in one or more storage servers. Clients can request this information from the metadata server and then access the storage nodes directly for actual I/O. The metadata server is also be responsible for locking blocks during write operations, as well as tracking versions for each object to avoid inconsistencies in the file system. It also monitors heartbeats sent by the storage nodes, adapting accordingly to the availability or unavailability of different machines in the network.

Currently there is only one metadata server in the cluster. We recognize this as a single point of failure, and intend to address this in the future by adding support for metadata server slaves that can replace the master metadata server in case of failure.

3 Design

3.1 File Structure

HFS maintains structure through a small database created and managed by the metadata server. Each data item is represented as an object with a distinct id. Object size, block size, current version are kept in the database.

In the prototype implementation, for each object we record the file name, type and the id of its parent object. We add a primary index containing parent id and filename, which effectively translates a pathname to its unique object id. Associated with each object id, a list of blocks is maintained, which specifies which block has a given range of bytes, and the names of the servers that are storing each block.

The database access object supports regular filesystem operations, such as path lookups, and file deletion/renaming. In the object initializer (constructor) we specify a secondary database to index entries based on the parent object and filename tuple.

3.2 The Metadata Server

The metadata server controls every function of HFS. It is the starting point for a client requesting storage, retrieval, modification and deletion of a file. It is

also a synchronization hub where all data servers announce their current status, synchronize their block lists, and report successful replication/write operations. The metadata server performs several functions as described in the remaining paragraphs of this section.

Data Server Management When launched, each data server announces itself to the metadata server. The metadata server keeps an internal record of data server names as well as the IP address and port the data server is listening for incoming connections. Also upon request of a data server, it will synchronize its block list with that server to verify that a given data server is storing the latest versions of the block it has on local storage.

If the data server is holding a stale block, the metadata server will tell it where can it get a more current version of the block. If the block has been deleted, then the metadata server instructs the data server to remove it from its local storage. This helps keep the filesystem in a consistent state among all of the live nodes.

File Creation When a client wants to create a new object, it goes to the metadata server to specify the type of object it wants to create. Along with the request, instructions are provided to specify whether the object should be split into blocks (and the preferred block size) and also how many replicas should the filesystem keep of each block.

This allows the filesystem to be configured on a per-file basis. For example a parallel application might require a certain block size which enables different clients to write their data units to a single object at the same time. Another application might not desire objects to be split in blocks, since the allocation units are small (or large) enough, where there is clearly no need for parallelism.

Upon successful completion of a request, if the metadata server grants creation of the object, an object id is passed back to the client along with the address of data servers where it can connect to create blocks.

Block Locking The metadata server keeps an internal data structure with a list of blocks which are currently locked by a write operation. When a client is attempting a write operation on a block, the data server serving the operation must first request a lock from the metadata server. If the lock is granted, further requests to modify that block coming from other servers are denied until the lock gets released on completion of the operation.

In our current implementation, locks timeout in the metadata server after a specified amount of time (in case the data server crashes). If the data server needs to keep the lock for a longer period of time, it should renew the lock with the metadata server.

Replication Whenever a block is created or a write operation on a block completes, the metadata server checks the replication policy established for the parent object of the block. It then instructs a given number of data servers to replicate the block until the policy is satisfied.

Load Balancing The metadata server tries to ensure that no data server gets overwhelmed with multiple requests in a short period of time, as this can affect IO dramatically. In this version of the file system, a Least Recently Used (LRU) policy is implemented, where the data server that was used more recently will be the last one on the queue whenever the metadata server has to select a storage node to perform an operation on.

The Data Server A data server acts primarily as a storage node for the distributed filesystem. Its operations are limited to reading and writing blocks, and informing the metadata server of its status. A small database is kept in each node, where the data server tracks the blocks it has along with their respective versions. Every time an object is created or modified in a data node, that node updates its database to reflect this change.

Consistency Whenever a data server is started or resumed (for example after a crash), it will contact the metadata server and request block synchronization. The data server will compare the versions of the blocks it has with the metadata server. In case there are any discrepancies, the metadata server will send instructions on how to correct this (either by deleting the block, or by retrieving the latest version from a different data server).

In addition, whenever a block is modified, all online data servers which store copies of the block are alerted of the version change by the metadata server. Using this information, data servers may redirect requests for a stale block to a server which has the updated version.

Read/Write Operations Read and write operations are performed on the blocks the data server has in its local storage. The data server checks against its object database to ensure it has the current version of the requested block. Should the data server have an older version, it will deny the request, and contact the metadata server to obtain an updated version of that block.

After a write or a create operation, the data server contacts the metadata server and registers its success, so that the metadata server can take appropriate actions. Such actions notify all of the data servers holding the block that a new version exists, and facilitate the replication process among them (according to the replication policy).

The client API supports random reads and writes (both within and between blocks) via a seek call.

3.3 Client Operations

Client operations are basic filesystem operations. Currently when the client opens a file, it contacts the metadata server, which gives back a set of basic file properties along with the list of blocks that compose the object and their respective locations. The client API stores these properties in an internal object, so it doesn't have to contact the metadata server each time it wants to read or write to a specific location of the file.

Fig. 2. A read operation

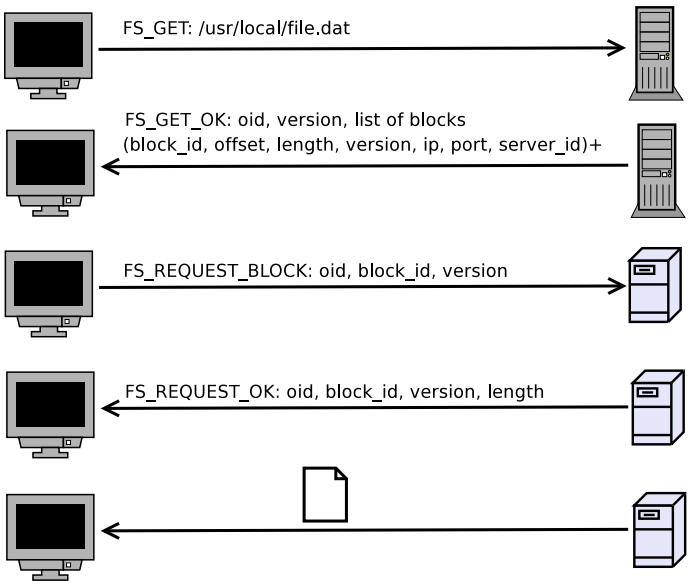


Figure 2 illustrates the life cycle of a read operation. The client first contacts the metadata server to request file data (including blocks, versions and locations). When the client actually wants to read a range of bytes, it must go to one of the data servers that have the given block and perform a GET_BLOCK request.

Fig. 3. A write operation

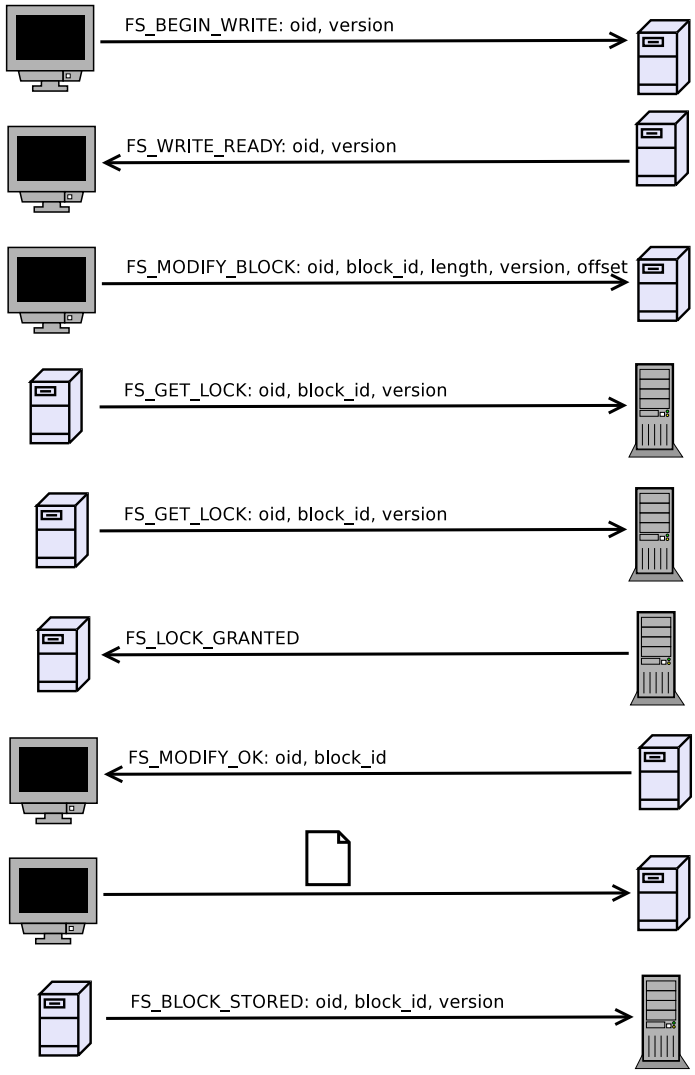


Figure 3 illustrates the life cycle of a write operation. Note that the client does not have to go to the metadata server again to request blocks, since this was done in Figure 2. Whenever the client wants to modify a block, the data server must request a lock from the metadata server before the operation can proceed.

Since each data server is notified whenever changes are performed in a block, the client can be confident that even if the block location in its cache list is outdated, the data server will be able to notify it if a newer version exists somewhere else.

3.4 Framework Structure

We have strived to make HFS as extensible and modifiable as possible in order to provide third parties with the opportunity of adding or changing functionality to the filesystem. We accommodate for changes in design via a module loading system—a well-known technique that has been used, notably, in the Linux kernel—wherein additional plug-ins are loaded at startup. Currently one can register two types of plug-ins:

Packet Builder One can register a class for building any type of packet. This allows one to change a packet's signature to add extra functionality to an operation. In a concrete example, perhaps we want to use different parameters for looking up a file with the FS_GET operation. The extra parameters can be added in a new class that specifies how to build this packet.

Packet Handler Different packet handlers can be added for any given packet. Through the use of a priority system, we can add several "layers" of classes that handle an incoming packet. For example, one could set up the functionality layer with a priority of 1, a log layer with a priority of 2, and implement an Observer pattern in a third layer that notifies different machines/processes of events happening to a given file. In a different example, one could completely replace the objects that handle lock request, so that a different lock policy can be implemented.

4 Implementation

HFS has been prototyped in the Python programming language, which has proven to be an outstanding choice for fleshing out the requirements space, especially the metadata component. As most of the runtime (especially for I/O) for critical libraries (such as I/O) is written in C, the part that is actually done in Python is not on the critical path. Much work in filesystems, especially lower-level approaches, is difficult to maintain and adapt for more modern applications and needs. In brief, the following tools have been employed in developing this first-generation and fully-working prototype.

Filesystem database The filesystem database for the metadata server was created using Berkeley DB [8], specifically through the bsddb3 library for python. For these tasks, BerkeleyDB was chosen because of its small size and speed, since the full power of a SQL server is not required. Berkeley DB is written entirely in C and C++ and is well-known for its performance.

The Communications Layer All transport and communications between nodes, metadata server and clients are done through TCP/IP using the C socket API that is available as is to Python programmers. A simple two-layer wire protocol was implemented. Although there are already approaches to do this (like XML-RPC or SOAP), a custom packet protocol was created to provide for faster I/O (requiring less message overhead and CPU processing time). The first layer of the protocol only describes packet length and type, and the rest of the message is passed to the appropriate packet handler (depending on the packet type) on the next layer of the protocol.

Fig. 4. A sample packet

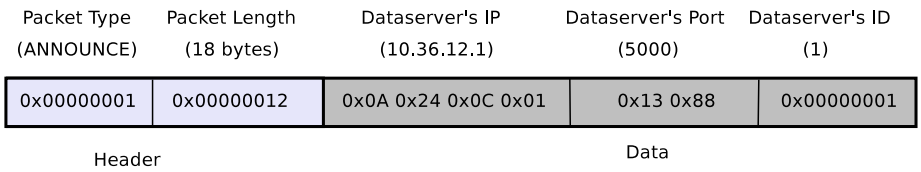


Figure 4 shows a sample announce packet used in the handshake protocol between the metadata and data servers. The header shows information on type and length, and the rest of the data is passed on to the appropriate handler in the metadata server.

The second layer is the equivalent of the presentation layer found in the oft-referenced OSI model. Its job is to transform the contents of the message into a structure that the application layer can understand and subsequently process. Python greatly facilitates this possibility by supporting structure packing (a technique employed in the implementation of many Internet services).

Configuration Files Configuration files are used for the metadata, data servers and the client library. These XML files are parsed when the server is first executed. Parameters for the data server include: connection backlog, server name, ip, port, addresses of possible metadata servers and storage path. The metadata server configuration file also includes default block size and number or replicas. For the client library the configuration file simply contains addresses of possible metadata servers, since this is always the starting point for a transaction.

Client API The client API was implemented using file-like objects in python. A user can instantiate a HFS API object and subsequently use it as a regular file, allowing for a transparent use in an application. One can also specify block size and number of replicas for a file in a straightforward manner. For example, a program for creating a file with block-size 65536 and then writing data to a file would be:

```
hf = HFSFile('conf_file.xml')
```

```
hf.setBlockSize(65536)
hf.open('/usr/local/large_file.dat', 'w')
hf.write(buf)
hf.close()
```

The file object took care of parsing the xml file, contacting the metadata server to satisfy the block request, and on the open call determine whether the file needed to be created or just opened for writing. If the file already existed, at this point an FS_GET request would have been sent to the metadata server to retrieve file information and block locations.

When the write method was invoked, a write session was opened with a dataserver containing the blocks in question (depending on the length of the buffer), and such blocks were modified or created.

5 Related Work

OceanStore [4] builds a distributed system focused on providing secure access to persistent information on a global scale. Freenet [3] deals with the same topic, but focuses on allowing publication, retrieval and replication of data while preserving the anonymity of authors, publishers and readers. Although our work touches on distributed filesystems, it focuses more on supporting parallel applications and data centers.

Similar in architecture to Lustre [7], we allow for dynamic block sizes—specified at the API level—to provide for parallel writes as well. We also support block & file versioning for consistency as it's done in the Sorrento [10] filesystem.

The Google File System [2] is one of the notable works on scalable storage, as it has been successfully implemented in a very large cluster. It provides fixed block size support for concurrent operations, and focuses on providing support for large blocks of data being read and written on a distributed network of commodity hardware. The Panasas ActiveStorage cluster [6] focuses on object-based storage and segment aggregation techniques to increase bandwidth and achieve scalability.

IBM's GPFS [9] uses byte-range locking to provide concurrent access to a file, while on our current implementation we opt for block locking providing for fixed data structures described at the application level. Note that the lock policy can be easily changed by specifying a new packet handler and even a new packet signature for the lock transactions as described in section 3.4.

6 Conclusions and Further Work

HFS demonstrates that it's possible to have a distributed object storage system that responds to different application requirements. By being able to split objects into custom block sizes, a parallel application can guarantee that all data units can be written to a file at the same time.

Currently there are no security considerations implemented in the HFS. A reasonable security implementation must be devised that does not significantly impact the overall performance of the file system.

Also the current implementation only provides one metadata server, which can be a single point of failure of the whole system. One way of solving this problem would be implementing a master metadata server with several slaves, which can become masters upon failure of the current metadata server.

We have explored using FUSE [5] as an interface between the kernel and HFS, thereby making it mountable to expose HFS as a regular filesystem under Linux. A prototype implementation using the FUSE Python bindings is available but is beyond the scope of this paper.

We are also exploring integration of this work with a scalable distributed lock server developed in [1] that supports byte-range (or block-range) locking.

7 Acknowledgments

This work was funded in part by a grant from the National Science Foundation to Loyola University Chicago (CCF-0444197) The authors would also like to acknowledge the advice and input of Michael Tobis of the University of Chicago Geophysics Department for his many helpful and insightful suggestions for improvement.

8 Bibliography

References

1. Peter Aarestad, Avery Ching, George K. Thiruvathukal, and Alok N. Choudhary. Scalable Approaches to MPI-IO Atomicity. In *Proceedings of Cluster Computing and the Grid (CCGrid)*, 2006.
2. Sanjay Chemawat, Howard Gobio, and Shun-Tak Leung. The Google File System. In *19th ACM Symposium on Operating Systems Principles*, 2003 October.
3. Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proceedings of the ICSI Workshop on Design Issues in Anonymity and Unobservability*, June 2000.
4. John Kubiatowicz et al. OceanStore: An Architecture for Global-Scale Persistent Storage. In *Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, November 2000.
5. Fuse: Filesystem in userspace. `http://fuse.sourceforge.net/`.
6. Abbie Matthews JDavid Nagle, Denis Serenyi. OceanStore: An Architecture for Global-Scale Persistent Storage. In *ACM/IEEE SC2004 Conference*, 2004 November.
7. Lustre. `http://www.lustre.org`.
8. M. Olson, K. Bostic, and M. Seltzer. Berkeley db. In *Proc. of USENIX Tech. Conf., FREENIX Track*, 1999.

9. Frank Schmuck and Roger Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proceedings of the FAST 2002 Conference on File and Storage Technologies*, San Jose, CA, January 2002. IBM Almaden Research Center.
10. Hong Tang, Aziz Gulbeden, Jingyu Zhou, William Strathearn, Tao Yang, and Lingkun Chu. A self-organizing storage cluster for parallel data-intensive applications. In *SC*, page 52, 2004.