Failure-Averse Highly-Available Distributed-File-System (FAHAD-FS)

Noah Epstein, Gabe Terrell

1. Introduction

FAHAD-FS (Failure Averse Highly Available Distributed File System) implements a distributed networked file system that shares files across a number of storage nodes to improve reliability and increase performance. The filesystem has three major components: 1) a filesystem client that a user runs on a host machine to perform standard filesystem operations (upload to/download from filesystem, delete, etc); 2) a “MasterNode” that manages access to and tracks names and locations of files within a distributed storage cluster; and 3) a cluster of storage nodes called “FileNodes” which store the filesystem’s data on their local storage.

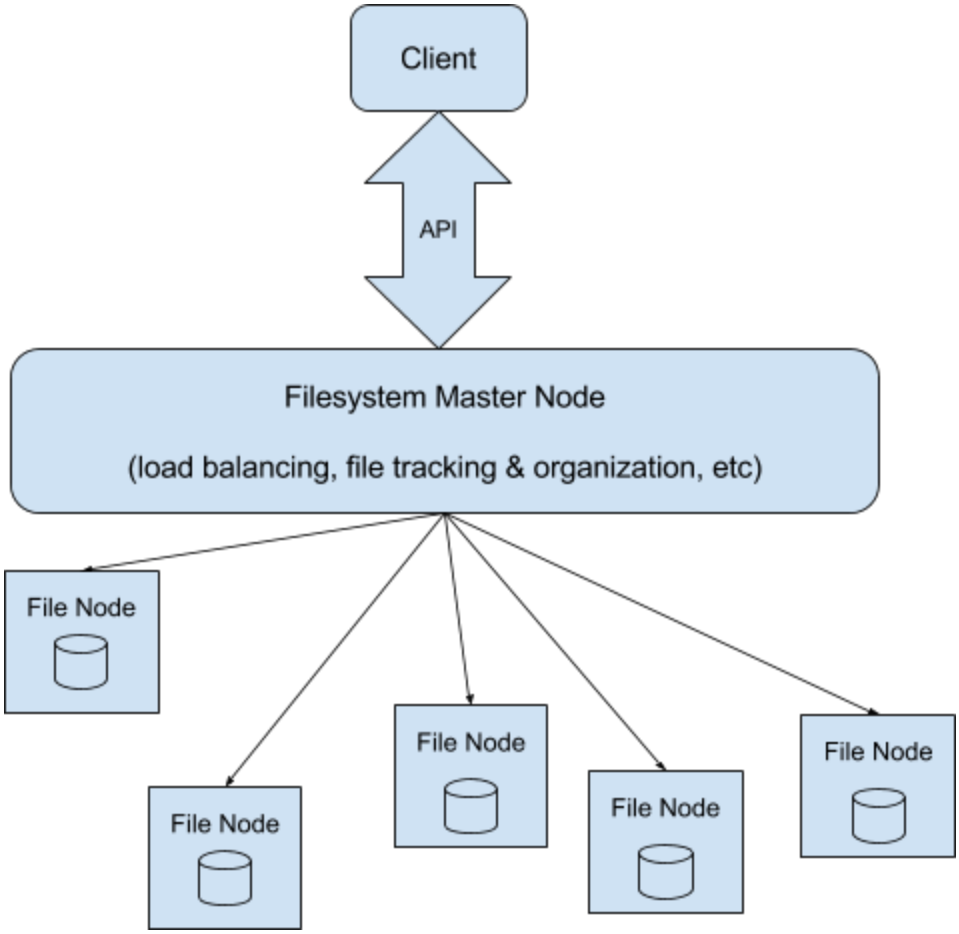
Distributed filesystems incorporate features to ensure resiliency and consistency of data. We drew inspiration from previous distributed filesystems such as the popular open source HadoopDFS [1], which employs a similar architecture to our filesystem, with a nameserver (analogous to our MasterNode) and a cluster of storage nodes; we also note Baidu’s more recent Baidu File System (BFS) [2], which implements filename tracking using a distributed Raft cluster, improving resiliency of the tracking server. FAHAD-FS includes features like: load balancing by the MasterNode so as not to overload any particular FileNode; “Recovery Mode”, where the MasterNode can dynamically react to the failure of file storage nodes by redistributing the failed node’s files to other nodes in the cluster; and multiple replications of files across nodes in a cluster.

2. Architecture & Features

The key design choice in FAHAD-FS is the separation of work between a central master node and a cluster of file nodes that respond to the master node.

FIle nodes are designed to simply store files. They can receive commands from the master node to download new files from a client, upload files to a client, or send copies of files to other file nodes.

The master node is responsible for management of the entire system, handling client requests, and handling file node updates. The master node does not actually store any file data other than a registry of all file names and paths in the system along with which file nodes own each file. The master node has two exposed ports: one to interface with clients and one to interface with new file nodes connecting to the cluster. When a client wants to interact with the file system, it will send a request to the master node which will parse the request and determine which file nodes the client needs to communicate with to finish the request. The master node also takes status data from file nodes and can respond accordingly, instructing file nodes to transfer data between each if needed to ensure file resiliency.



Client API

With a master node running and connected to a cluster of file nodes, the client has two primary forms of interacting with the server:

1. File Viewer: (client.py -v)

   The master node utilizes a tree-based directory data structure that is built at runtime from the file registry, and allows for a client to remotely interact with the file system, very similarly to the way a user may interact with their own files through a terminal system. Commands include the familiar ‘ls, cd, mkdir, pwd’. While the file viewer is a helpful tool to see all of the files on the system, it was not the main focus of the project, and is mostly used as a simple interface.

2. Upload/Download (client.py -u & client.py -d)

   The client can upload files from its local system to the remote server and download files from the server to its local system by sending requests to the master node’s exposed client port. While no actual files are stored in the master node that the client

communicates with, the master node will process upload and download requests by delegating them to the proper file nodes.

FAHAD-FS handles upload queries with the goal of load balancing. The master node maintains a priority queue of all online nodes and the current amount of storage and activity the nodes are receiving. When a new file is to be uploaded, the master node will utilize the priority queue to find the nodes with the most amount of open space. The master will then create a new upload session to keep track of which files should be receiving the data before sending these addresses to the client. As the client sends the data to each file node, the file node will verify receipt of the file using a checksum. The master will verify receipt of the file before logging the node into the registry. If a node fails to receive the file properly or dies before receiving the file, the master can respond to this by editing the session data and alerting the client to respond appropriately. This three-way form of communication keeps the master in charge of orchestrating completion of the task while delegating all of the computationally heavy work to the client and file nodes.

Likewise with downloads, FAHAD-FS will point the client to an available file node that has a copy of the desired file. Through the delegation patterns, the master node can handle a large number of requests by quickly determining which file nodes to delegate the download/upload work to. Using sessions and three-way communication, the master node can verify that the request has been fulfilled without doing any of the heavy-lifting.

Consistency Via Sessions

FAHAD-FS employs a session-based access procedure for file-accesses that require interaction with file nodes (upload, download, deletion, etc). Once the Master Node registers that a file will be accessed on the file nodes, it opens a session for that file. Any clients (other than the initial client) who attempt to access the file while it undergoes modification are required to wait until the session is closed to continue with any modifications they wish to make. This session-based architecture ensures that all clients who access the filesystem access particular files sequentially, eliminating the possibility that multiple clients modify a file concurrently. This, along with the three-way request-act-verify procedure, ensures that the files stored on any online nodes are consistent with the latest version modified by a client.

Status Checking

Every 5 seconds (or some configured period), the master node will send out a ping message to all online server nodes that includes a list of all file names stored in the registry along with each file's associated checksum. The file node can then use this file list to bookkeep the files it has stored to disk. The file node will ensure that all of the files are still in the list (i.e. weren't remotely deleted by a client) and if the files stored are still up-to-date (a file node could receive a file, then go offline when a new version of that file was uploaded; if this is the case, the checksum would not match the file stored on disk, and the file node would realize its copy of the file is obsolete).

In performing these status checks, the filenode will ensure that all files it has stored on disk are non-obsolete and will prevent wasted disk space.

Once the filenode has completed its bookkeeping, it will respond to the master node's ping message with a list of recent activity the node has done along with the current amount of disk space the node is currently using to store its files. Once the master node has received replies from all of the file nodes (or the sockets timeout if file node is dead), the master node can respond to the new status of the system. With an up-to-date record of all active file node's disk space usages and activity, the master node can update its priority queue to ensure the nodes with the least amount of storage consumed and activity will be next to receive new files.

Additionally, if the master node realizes that a file node has died since the last status check, it can enact Recovery Mode.

Recovery Mode
Recovery mode is the master node's protocol to respond to node loss to ensure file resiliency. The master node's goal is to ensure that there is always k copies of each file in the online system (k = 3 for our current version of FAHAD-FS). When one or more nodes is lost, the master node will scan through its file registry and determine how many online nodes have a copy of each file on record. If that number is less than k, the master node will use its priority queue to determine which file nodes should receive copies of the file from the other file nodes that still have that file on record and send the proper messages between file nodes to conduct the file copying process. This is accomplished in real time without any interaction of a client.

Through Status Checking and Recovery Mode, FAHAD-FS can ensure availability of all files even in cases of sporadic node loss.

3. Performance Results & Testing

We used an automated bash script to test the resiliency and load impacts of FAHAD-FS. The script was capable or randomly adding and deleting file nodes from the cluster while simultaneously sending a number of clients randomly requesting uploads and downloads of various files. Within the limits of working on a single laptop computer, the largest test conducted thus far hosted 30 initial file nodes that served up to 10 upload and download requests per second span while nodes randomly entered and exited the server, with some old nodes returning and some completely fresh nodes entering the cluster. The system showed demonstrated resiliency and node balancing. By the end of the testing, the master node and file nodes that weren't killed off remained standing, and the master node demonstrated that the distribution of storage across nodes on the system was fairly balanced.

As we increased the number of nodes and requests being served per second, we ran into some concurrency issues in the master registry which likely stemmed from oversight on our parts about non-atomic functions (i.e. race conditions from working with a multi-threaded server). This

is very likely to be fixed if we were smarter about the use of locks and mutexes in key functions of the server. However, in low-stress environments, the system performed as we expected.

4. Further Work

a) **Client API Buildout:** Some standard filesystem commands remain unimplemented on FAHAD-FS (‘CP’, ‘RMDIR’, ‘MV’) because they would not help us demonstrate the performance of the filesystem. However, for the filesystem to be useful for users as more than a demonstration, these must be added to the client API.

b) **Chunk-Based Data Storage:** The filesystem currently stores an entire file on a FileNode’s local storage, so when the client uploads or downloads a file, they must transmit the contents of the entire file to each FileNode who stores it. However, to more appropriately distribute load across the storage cluster, we would like to implement storage in “chunks” of a fixed maximum size, so that very large files may be stored in pieces. This will allow the MasterNode to distribute the load of uploads, downloads, and storage such that fewer nodes have choked bandwidth or run out of local storage due to the access of a single large file.

c) **Improve Load Balancing Heuristics:** We would also like to perform load balancing using a more accurate heuristic to characterize load. FAHAD-FS exclusively uses proportion of allowable disk quota to determine load. A more accurate representation of load would take into account available FileNode bandwidth, request rate, CPU usage and RAM usage as well.

d) **Client-Side Caching:** Many popular distributed filesystems employ some form of caching using the filesystem on a client’s host machine for recently accessed files. While this can affect consistency of files stored within the system (i.e. clients may perform work on a cached but stale version of a file), implementing client-side caching would allow significantly improved filesystem performance and user experience, especially for larger files.

e) **Multiple Authenticated Users:** We could also implement support for multiple authenticated users who can use permissions to access subsets of the filesystem. This would more accurately reflect a real-world filesystem.

f) **Improved Testing for Latency and Concurrency Issues:** We would also like to expand the testing infrastructure used with FAHAD-FS. We encountered issues with race conditions when testing with large numbers of concurrent clients and filenodes, so further testing would allow us to improve support for many concurrent clients. Also, we only tested on localhost and on low-latency links within private networks at Tufts. Further testing could be conducted over the public internet, which would illustrate performance of the filesystem under “real” network conditions.

There are still many features of FAHAD-FS that can be improved, and even more that could be added in future iterations. However, we are very satisfied with the performance of FAHAD-FS. It was a fun and challenging project that allowed us to gain experience with many aspects of networking and distributed systems!

Additional Reading

[1] Hadoop Distributed Filesystem: http://ieeexplore.ieee.org/document/5496972/?arnumber=5496972&tag=1

[2] Baidu Filesystem (BFS): https://github.com/baidu/bfs

[3] Andrew Distributed Filesystem: https://en.wikipedia.org/wiki/Andrew_File_System