JFusionFS

A Java Implementation of FusionFS

Eric Faurie
Dept. of Computer Science
Illinois Institute of Technology
Chicago, USA
efaurie@hawk.iit.edu

Chaitanya Reddy Chatla
Dept. of Computer Science
Illinois Institute of Technology
Chicago, USA
cchatla@hawk.iit.edu

***Abstract* – FusionFS is a node local distributed storage system that was developed for High Performance Computing systems. FusionFS disperses metadata to all available compute nodes through the use of a distributed hash table (DHT), and thus overcomes the metadata problem common in many storage systems. FusionFS also relies on a parallel file system (PFS) which acts as a large file store (LFS) when the files cannot fit on local devices. We propose to implement a DHT and LFS agnostic implementation of FusionFS in the Java programming language.**

***Index Terms* – FusionFS, HPC, metadata, distributed file system**

I. Background Information

The existing architecture of high performance computing systems is decades old and has its compute and storage resources separated. This architecture is no longer feasible for modern data intensive applications which require a lot of support from storage subsystems [6].

FusionFS is a distributed file system which has been designed to store large volumes of data reliably and stream those datasets at high bandwidth to user applications [5]. FusionFS stores metadata and application data separately. It stores metadata in a distributed hash table and application data on node local disks or a parallel file system. The distributed hash table allows metadata availability on all active compute nodes and thus achieves maximal concurrency of metadata operations. Because FusionFS is implemented in C, and written for the HPC environment, it supports legacy code through a POSIX interface. It implements this interface through the FUSE libraries.

We propose the implementation of FusionFS in the Java environment in such a way as to increase its scope of applicability, ease of use, and flexibility.

II. Problem Statement

Large scale experimentation generates huge amounts of data in both size and number. Typically clusters store these files in some sort of central persistent storage. This not only introduces a single point of failure, but also burdens the network with unnecessary traffic.

In HPC applications where network bandwidth is a commodity and datasets are considerably large, moving data becomes unfeasible. FusionFS is a distributed file system designed to tackle this problem. It has a proven ability to increase performance by persisting data to local compute storage and greatly reduce network traffic.

The widespread use of commodity cluster, cloud, and distributed computing is increasing rapidly among hobbyists and those in industry. This has been, in some part, caused by the development of easy-to-deploy data management and analysis stacks such as Hadoop. However, these applications suffer from many of the same problems that the HPC environment does, that is, the separation of data and compute, or the aggregation of metadata on a single server.

These platforms would benefit from the use of FusionFS, however, FusionFS is a C implementation written with the HPC environment in mind. If one would like to increase the performance of a Hadoop deployment with FusionFS, an adapter would have to be written. Besides the adapter, FusionFS is also tightly coupled to POSIX, FUSE, ZHT (a distributed hash table implementation), and a Parallel File System. This means that if one were to use FusionFS they must also use ZHT, FUSE, and Parallel File Systems, and thus they cannot seamlessly integrate FusionFS into an existing system. This is especially problematic in industry deployments where certain application or infrastructure restrictions may exist.

III. Proposed Solution

We propose a Java native implementation of FusionFS called JFusionFS. Our goal is to provide a dynamic implementation of FusionFS that is dependency agnostic. That is, it allows users to make infrastructure decisions while supplying the FusionFS framework for file management. This implementation is designed with as few environment assumptions as possible, which means it is designed with few dependencies and architected in a modular way.

A Java native implementation offers multiple benefits. It can natively serve as a replacement to the Hadoop Distributed File System without an adapter. It can be easily deployed on various types of commodity clusters, whether that be in the cloud or otherwise. Increasing FusionFS's ease of deployment means we widen its applicable user set.

A modular architecture allows us to implement the FusionFS file management framework without making assumptions concerning underlying distributed hash tables and large file stores (LFS). This will allow users to decide on the underlying metadata manager and create an LFS that effectively utilizes existing resources. This will increase its appeal to industry users that may have application or infrastructure requirements.

IV. Architecture and Implementation

The architecture of FusionFS can be divided into two general components, File Management and File Transfer. The following sections will discuss the responsibilities of these two components, how they interact, and how they are realized in code.

The entire JFusionFS implementation is about 1000 lines of Java code. This includes the Amazon S3 LFS and the Amazon DynamoDB DHT interfaces.

Interestingly all of JFusionFS's core is developed using standard Java Libraries. This means that only vanilla java is required to use JFusionFS and it requires the install of no other packages. This eases deployment greatly. That being said, the DHT and LFS implementations will most likely require a third party library. In the case of the Amazon Cloud backend, the Amazon AWS SDK is the only other dependency [2].

All of the code was developed using Eclipse with the Amazon AWS SDK Toolkit and the code was version controlled with a BitBucket hosted Git repository [7].

Sections *A* and *B* describe the overall architecture, and the following sections describe other important components in detail.

A. File Management

The File Management component is responsible for handling local file I/O, interaction with the LFS, metadata management, and handling data locality decisions.

Perhaps most importantly, the data management component handles metadata operations. It does this by interacting with the Distributed Hash Table (DHT), or key value store.

The File Management component is composed of seven components: the JFusionFS Instance, Dynamic Class Factory, Distributed Hash Table, Large File Store, JFusion File Reader, JFusion File Writer, and the JFusion Config Reader.

The JFusionFS Instance is the exposed API of JFusionFS and will be discussed in more detail below.

The Dynamic Class Factory is the plugin handler. That is, it is a class which utilizes Java generics and Java Reflection to dynamically load a specified class from within the class path and instantiate the class as an object that is returned to the JFusionFS Instance [3,4]. This means that we can allow the user to configure files that will be used as the DHT and LFS without changing the JFusionFS codebase.

The Distributed Hash Table and Large File Store are interfaces that are implemented by adapters written by the user. These interfaces are discussed further below.

The JFusion File Reader and Writer are the low level logic concerning IO. They handle all of the metadata operations, file access, and network usage.

The JFusion Config Reader is a custom config reader designed to allow important fields to be accessed easily and in a readable way.

If the file is to be written, opened, or read locally, the IO is handled within the File Management layers using typical Java File IO calls. This simplifies the stack and reduces the number of dependencies. If the file is to be written, opened, or read from the LFS, an LFS adapter that implements the JFusionFS LFS interface is used.

If the file is to be read from a remote node, the File Management layer will submit a request to the File Transfer layer.

B. File Transfer

The File Transfer layer is a component used exclusively for providing access to files on remote nodes. A JFusionFS daemon process running on each participating node listens for incoming requests and when a request is received, transfers the requested file to the destination node.

This is achieved through the use of standard Java Net sockets and the TCP protocol for guaranteed data delivery. These connections are temporal, that is, the TCP connection is only kept open for a single request. This reduces the number of idle network connections and thus reduces unnecessary traffic.

Obviously, remote file requests must be processed asynchronously and thus the file transfer daemon is designed with multithreading. The daemon spins off individual connection threads to handle the transfers.

The File Transfer component is composed of three modules: the File Transfer Daemon, File Transfer Thread, and File Transfer Client.

The file transfer daemon is a simple server process which runs as a service on the participating node. This process listens on a configurable socket for incoming connections.

Once a connection is received, the File Transfer Daemon will spin off a new File Transfer Thread which handles the interaction with the requesting node. This thread will receive commands from the requesting node and transmit the locally stored file back to it.

The File Transfer Client is the requesting nodes interface. When the JFusionFS instance determines that a file is not stored locally or within the LFS, it will use an instance of the File Transfer Client to connect to the remote node's File Transfer Daemon to request it.

C. Interfaces

JFusionFS was developed with a plugin infrastructure. The goal of the design was to allow users to decide how to implement the underlying dependencies and to reduce assumptions concerning the deployed environment. These configurable dependencies are the Distributed Hash Table, and the Large File Store.

The implementation of both the adaptors for DHT and LFS can be done without changing the JFusionFS code. The user simply writes an adapter class that implements the included *DistributedHashTable* or *LargeFileStore* java interfaces, includes the compiled code in the classpath, and inserts the name of these classes in the JFusionFS configuration file. These classes will then be loaded and instantiated by the Dynamic Class Factory mentioned in the Data Management section above.

Both the distributed hash table and large file store will be described in detail below.

Distributed Hash Table

The original FusionFS implementation uses a distributed hash table called ZHT. While ZHT is a great fit for FusionFS and is highly optimized, the goal of JFusionFS was to maintain flexibility and reduce dependency coupling. It should be noted, that ZHT can still be used in JFusionFS if an adapter was written.

The Distributed Hash Table java interface requires only two operations, put and get. In our case study we implement a Distributed Hash Table using a key-value store hosted in Amazon's DynamoDB. The adapter for this DynamoDB DHT is just 20 lines of code.

This plugin gives the user the power to decide how the underlying DHT is implemented. It also allows FusionFS to be easily integrated into existing systems that may already use Distributed Hash Tables.

Large File Store

FusionFS assumes that a parallel file system exists in the environment. JFusionFS aims to allow the handling of very large files without the assumption of a parallel file system. This is why this adapter is named the large file store. It acts as an overflow when the file cannot be stored on the local node's disk. It can be implemented any way the user sees fit by writing an adapter that implements the JFusionFS *LargeFileStore* interface. Only three functions are required, write, open, and read.

Realize that the large file store can be implemented using a parallel file system, in which case the adapter would simply reroute requests to the parallel mounted volume and essentially use standard Java IO calls. However, we don't assume this architecture. Also realize that if one chooses, an adapter can be written that routes these files to a predefined node with vast amounts of disk storage using TCP. If this is done, it is essentially no different than the remote read and in fact can even be written using the underlying JFusionFS file read code. However, we will still route through the adapter code when the LFS is used because we do not assume this naïve implementation.

In our case study, we implement the LFS using the Amazon S3 cloud storage system. This is done in ~40 lines of code and thus demonstrates the simplicity of such an implementation. The LFS being used can easily be changed by simply changing the configuration of JFusionFS. No code changes within the system are required.

D. JFusionFS API

The JFusionFS API is kept as similar to the FusionFS API as possible while keeping within typical Java naming conventions. The API is defined in the Java Object *JFusionFSInstance* and these calls are *ffsOpen(), ffsRead()*, and *ffsWrite()*. For the sake of usability JFusionFS also includes an *ffsAppend()* operation. The append operation will become useful if JFusionFS is used to enhance the Hadoop stack. Note that in the Java implementation *ffsClose()* is not included as the Java garbage collector will automatically destroy file descriptors when they are no longer used.

These operations behave much as expected. The *ffsOpen()* call returns a Java File object, while *ffsRead()* returns a *BufferedInputStream*. *ffsWrite()* and *ffsAppend()* simply outputs provided data from a byte array or an output stream.

Note that these calls only require a file path and data to be written (if a write call). All metadata, locality, and retrieval operations are handled in the background.

For JFusionFS to be used, unfortunately application code will have to be changed. However, the changes are straightforward.

Simply changing the individual file IO lines from the Java File IO library to JFusionFS will suffice.

V. Flow of Control

This section will describe logical decisions throughout execution. More specifically, how the JFusionFS instance handles a typical read or write call.

A. *ffsRead()*

When a user reads a file, it first looks up the desired file in the DHT. If the file exists, various metadata will be returned. This metadata contains the location of the file as well as other file information. The location of each file may be one of three locations, the local node, a remote node, or the large file store (LFS).

Once the location of the file is determined JFusionFS will read the file. Obviously if the file is stored locally it simply opens the file and returns a BufferedInputStream where the data can be accessed. However, if the file is on a remote node, it uses the File Transfer Client to first pull the file and write it to the local disk before opening it and returning the stream. If it's stored on the LFS it uses the dynamically loaded LargeFileStore to read the file and return a stream.

B. *ffsOpen()*

The open command follows the same exact logic pattern as the *ffsRead()*. However, instead of returning a data stream, it returns the file handle itself.

C. *ffsWrite()*

When a file write call is made, the File Management component determines whether the file should be stored to local disk or stored in the LFS. The LFS is used for files deemed too large for the local disk either because the free space on the local node is less than required or the size of the file is above a file size threshold defined in the JFusionFS configuration file. The file manager, in this way, handles data locality decisions while writing.

Obviously if the file is to be written locally the JFusionFS File Writer writes it to the workspace with the specified path using standard Java File IO calls. If it is to be written to the Large File Store it uses the Dynamically Loaded LargeFileStore.

After the file is written, the DHT is updated with the file path, location (the local node's IP, or 'LFS' if stored in the LFS), and metadata. This is all handled by the File Management layers.

Note that this simple flow is always the same. If a user wishes to overwrite a file in JFusionFS a simple write will suffice. Because the DHT is updated only after the file is completely written all subsequent reads will request the latest file stored on the new nodes system even if the host of the latest file has changed.

D. *ffsAppend()*

The append operation is a little more interesting than a simple write. This is because the file being appended may not reside on the local system. To circumvent this problem, we internally make an ffsRead() call which will guarantee that the latest file is pulled to the local node. Once this occurs, the data is appended and the DHT is updated with the local nodes address so subsequent reads acquire the latest information.

VI. Deployment

A. *Choose a DHT and LFS*

The first stage of deployment is choosing a DHT and an LFS to use. Note that the use case adapters for the Amazon DynamoDB implementation of a DHT and the Amazon S3 store of the LFS are included in the JFusionFS code. If these will suffice than no additional adapters are required.

However, if the user wishes to use, for example, a CleverSafe system for the LFS a simple adapter can be written.

Once an adapter is chosen for each, simply insert the name of the adapter into the config file under the distributed hash table and large file store fields.

B. *Define Your Workspace*

Define a workspace within the config file. This workspace is the location on the local node where all files will be stored. Note that all file paths passed to JFusionFS are relative to this location.

C. *Start the File Transfer Daemon*

Start the file transfer daemon by executing the JFusionFS executable. This can be ran with or without a GUI. The simple GUI displays node information such as the disk space in use, the local hostname, the JFusionFS version in place, and a console window that displays the log.

Once this service is running the node is ready for incoming network requests for files.

D. *Run Your Application*

Run the custom application that writes, reads, and opens files using the JFusionFS API.

VII. Use Case – Evaluation

The use case for JFusionFS is a simple benchmark program that writes randomly generated binary files of a given size. The backend of JFusionFS in our use case, is implemented with the Amazon DynamoDB based Distributed Hash Table, and an Amazon S3 Large File Store. The adapters for these two interfaces are dynamically loaded using the JFusionFS configuration.

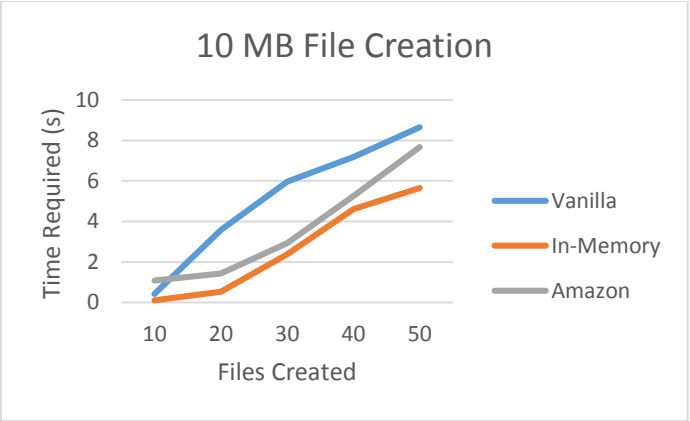
We chose the Amazon cloud platform backend because of its accessibility. Also, storing the DHT and LFS on an internet accessible cloud gives us some interesting benefits. This means that JFusionFS can essentially be used as a distributed file storage system in a platform similar to SETI At Home. That is, heterogeneous systems scattered throughout the world can all store files in a shared JFusionFS instance and communicate as if they were in a single rack cluster.

This being said, JFusionFS cannot be effectively compared to FusionFS without an implementation of a ZHT adapter and a parallel file system. However, we did test JFusionFS against typical Java IO operations to ensure that the added layers from JFusionFS don't throttle local IO.

We compare the use of JFusionFS with the Amazon backend, to an instance of JFusionFS with a dummy in-memory metadata table, and a small program using vanilla Java File IO calls. The results are as follows. The values in the tables below are the time required to complete the creation of all files in seconds.

1 KB Files			
Files Created	Vanilla	In-Memory	Amazon
10	0.000514	0.00585	1.02955
50	0.00694	0.01157	1.59134
100	0.006681	0.02183	1.91594
500	0.032554	0.09117	272.4555
1000	0.086502	0.14197	765.3909

10 MB Files			
Files Created	Vanilla	In-Memory	Amazon
10	0.41824	0.10565	1.08924
20	3.58339	0.53326	1.4364
30	5.97167	2.38985	2.93544
40	7.17528	4.60932	5.25945
50	8.65251	5.6584	7.66778



One can see when comparing the creation of 1KB files, that the In-Memory dummy DHT adds little overhead over the Vanilla Java file creation. The Amazon implementation does introduce a sizeable delay especially when file creation volume is high. However, this is mostly due to Amazon DynamoDB requests being throttled. One can raise the cost of the Amazon DynamoDB instance and increase the thresholds if required. This is also evidence that the performance of JFusionFS is greatly dependent on the implementation of the DHT and LFS. If a true Distributed Hash Table, such as ZHT were implemented, it will increase the speed over the DynamoDB instance.

Interestingly when creating larger files, JFusionFS tends to perform better than Vanilla java calls with both the in-memory and Amazon implementations.

VIII. Related Work

The idea of distributed metadata has been around for a while. Fault datacenter storage (FDS) is a high performance, fault tolerant, large scale, locality oblivious blob store which maintains a lightweight metadata server and offloads the metadata to available nodes in a distributed manner [6]. Salus is a block store that seeks to maximize simultaneously both scalability and robustness. It provides strong end to end correct guarantees despite a wide range of server failures [11]. It collocates compute and storage resources.

A file system which is similar to our version of JFusionFS is XtreemFS. XtreemFS is an object-based, distributed file system which has been written in Java. It has full and real fault tolerance while maintaining POSIX file system semantics [10]. We discard POSIX compliance but introduce a plugin architecture where the backend handling of metadata is customizable.

While these systems apply a general rule to deal with data I/O, FusionFS is optimized for write-intensive workloads that are particularly important for HPC systems [5].

IX. Future Work

In the future, we would like to implement a full ZHT adapter and write a benchmark application that can run on both JFusionFS and FusionFS. This would allow us to truly compare the performance of the Java and C implementations.

We would also like to write a Hadoop Interface and use JFusionFS as a replacement for HDFS. We can benchmark typical Hadoop applications with the HDFS backend and JFusionFS using various Distributed Hash Table implementations.

X. Conclusion

The goal of JFusionFS was to translate FusionFS into an extensible Java Framework that is modular and dynamic. The results of our case study show that most of the performance impact is dependent on the implementation of the DHT. However, because the framework works as expected, we can now explore various DHT implementations and push the boundaries of distributed Java IO. The resulting JFusionFS implements the ideals of FusionFS in a way that opens up its range of applicability and greatly increases its ease of use.

REFERENCES

[1] Amazon EC2, http://en.wikipedia.org/wiki/Amazon_Elastic_Compute_Cloud

[2] Amazon AWS SDK, http://aws.amazon.com/sdk-for-java/

[3] Bellia, Marco, and M. Eugenia Occhiuto. "Higher order programming through Java reflection." *Concurrency, Specification and Programming CS&P* 3 (2004): 447-459.

[4] Chiba, Shigeru. "Load-time structural reflection in Java." *ECOOP 2000—Object-Oriented Programming*. Springer Berlin Heidelberg, 2000. 313-336.

[5] Dongfang Zhao, Zhao Zhang, Xiaobing Zhou, Tonglin Li, Ke Wang, Dries Kimpe, Philip Carns, Robert Ross, and Ioan Raicu. "FusionFS: Towards Supporting Data-Intensive Scientific Applications on Extreme-Scale HighPerformance Computing Systems", IEEE International Conference on Big Data, 2014.

[6] E. B. Nightingale, J. Elson, J. Fan, O. Hofmann, J. Howell, and Y. Suzue,"Flat datacenter storage," in Proceedings of USENIX Symposium on Operating Systems Design and Implementation, 2012.

[7] JFusionFS, https://bitbucket.org/efaurie/jfusionfs

[8] P. Freeman, D. Crawford, S. Kim, and J. Munoz, "Cyberinfrastructure for science and engineering: Promises and challenges," Proceedings of the IEEE, vol. 93, no. 3, 2005.

[9] Shvachko, Konstantin and Kuang, Hairong and Radia, Sanjay and Chansler, Robert. The Hadoop Distributed File System, IEEE 26th Symposium on Mass Storage Systems and Technologies, 2000.

[10] XtreemFS, http://en.wikipedia.org/wiki/XtreemFS

[11] Y. Wang, M. Kapritsos, Z. Ren, P. Mahajan, J. Kirubanandam, L. Alvisi, and M. Dahlin, "Robustness in the salus scalable block store," in Proceedings of USENIX conference on Networked Systems Design and Implementation, 2013.

Appendix

Work was broken down into two general parts. The first part was implementation of the Use Case and exploring JFusionFS's coexistence with existing DHTs and LFSs. The second part was the implementation of JFusionFS itself.

Eric Faurie had part two and implemented JFusionFS, wrote the interfaces, and implemented dummy DHT and LFS plugins. He later assisted Chaitanya in implementing the Amazon Cloud based adapters for a DHT and LFS using Amazon AWS tools, and helped deploy the system on Amazon EC2 Micro instances for some simple benchmark tests that Eric wrote as well.

Chaitanya was in charge of part one, that is, looking into connecting JFusionFS with existing DHT and LFS instances as well as working on an API for JFusionFS to act as a replacement for Hadoop HDFS.