# DECAFS: A MODULAR DISTRIBUTED FILE SYSTEM TO FACILITATE DISTRIBUTED SYSTEMS EDUCATION

A Thesis

presented to

the Faculty of California Polytechnic State University

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Halli Meth

June 2014

COMMITTEE MEMBERSHIP

| | |
|---|---|
| TITLE: | DecaFS: A Modular Distributed File System to Facilitate Distributed Systems Education |
| AUTHOR: | Halli Meth |
| DATE SUBMITTED: | June 2014 |
| COMMITTEE CHAIR: | Associate Professor Chris Lupo, Ph.D., Department of Computer Science |
| COMMITTEE MEMBER: | Professor Alexander Dekhtyar, Ph.D., Department of Computer Science |
| COMMITTEE MEMBER: | Associate Professor Aaron Keen, Ph.D., Department of Computer Science |
| COMMITTEE MEMBER: | Associate Professor John Bellardo, Ph.D., Department of Computer Science |

ABSTRACT

DecaFS: A Modular Distributed File System to Facilitate Distributed Systems Education

Halli Meth

Data quantity, speed requirements, reliability constraints, and other factors encourage industry developers to build distributed systems and use distributed services. Software engineers are therefore exposed to distributed systems and services daily in the workplace. However, distributed computing is hard to teach in Computer Science courses due to the complexity distribution brings to all problem spaces. This presents a gap in education where students may not fully understand the challenges introduced with distributed systems. Teaching students distributed concepts would help better prepare them for industry development work.

DecaFS, Distributed Educational Component Adaptable File System, is a modular distributed file system designed for educational use. The goal of the system is to teach distributed computing concepts to undergraduate and graduate level students by allowing them to develop small, digestible portions of the system. The system is broken up into layers, and each layer is broken up into modules so that students can build or modify different components in small, assignment-sized portions. Students can replace modules or entire layers by following the DecaFS APIs and recompiling the system. This allows the behavior of the DFS (Distributed File System) to change based on student implementation, while providing base functionality for students to work from.

Our implementation includes a code base of core DecaFS Modules that students can work from and basic implementations of non-core DecaFS Modules. Our basic non-core modules can be modified to implement more complex distribution techniques

without modifying core modules. We have shown the feasibility of developing a modular DFS, while adhering to requirements such as configurable sizes (file, stripe, chunk) and support of multiple data replication strategies.

# ACKNOWLEDGMENTS

Thanks to:

- Chris Lupo and Alex Dekhtyar for project inspiration, support and guidance.
- Jeffrey Forrester #1 lab partner since 2010.
- Peter Faiman compilation magician, git wizard, and data storage sorcerer.
- My Mom and Dad for constant love and support (and letting me vent through my stress).

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

# CHAPTER 1

# Introduction

## 1.1 Distributed Systems in Education

Computer Science education is structured to be attractive to students and "strategic to a business enterprise [6]." Hogansan explains that curricula needs to be structured in a way that can counter the movement to out-source development work by preparing students in "knowledge areas that are of strategic importance to the enterprise" and are therefore "less likely to be successfully out-sourced [6]." His "strategic CS program" within the ABET criteria, includes Distributed Technologies at both the Undergraduate and Graduate level [6]. The importance of distributed computing in education can also be seen by Google and IBM's joint Initiative in 2008 to "improve computer science students knowledge ... to better address the emerging paradigm of large-scale distributed computing [13]."

The Cal Poly Computer Science Department has decided to begin updating the curriculum to facilitate Distributed Systems Education. This thesis aims to help shape coursework to teach Distributed Concepts by allowing students to write components of Distributed File Systems.

## 1.2 Distributed File Systems

Distributed File Systems have similar operational goals as non-distributed file systems such as basic file and directory manipulation (open, read, write, delete). However, a

Distributed File System (DFS) is "a file system, whose clients, servers, and storage devices are dispersed among the machines of a distributed system." Under this definition, a distributed system is "a collection of loosely coupled machines", servers are software services that run on a single machine in the system and client's are processes that can invoke these services [7].

A DFS can be implemented as part of a Distributed Operating System, or as a software layer [7]. At Cal Poly, we are implementing a software layer DFS that will be responsible for managing the communication between the conventional operating systems and file systems.

## 1.3 Our Contribution

Research in Distributed Computing Education has focused on virtual systems [23]. However, at Cal Poly the theme of each department's educational goals is "learn by doing". To adhere to the Cal Poly motto, we developed DecaFS, a modular distributed file system to run on physical systems.

Our work describes the following contributions:

1. Design of a modular DFS (DecaFS), with small components that can be implemented or adapted in a classroom environment.

2. Implementation of base functionality in DecaFS to allow students to develop DecaFS modules.

Another main component of this project is describing sample projects for students to complete within the DecaFS infrastructure. Design of lab activities for students to complete using DecaFS to learn various distributed computing concepts is described in my colleague's work [4].

Background information on Distributed File Systems and Related Work are detailed in CHAPTER 2 and CHAPTER 3. System requirements are detailed in CHAPTER 4. Design of our system, DecaFS, is presented in CHAPTER 5, followed by system workflows (CHAPTER 6) and implementation details in CHAPTER 7. Information on DecaFS testing and validation is presented in CHAPTER 8, and lastly Conclusions and Future Work in CHAPTER 9 and CHAPTER 10.

# CHAPTER 2

# Background

Distributed File Systems have many of the same goals of traditional file systems. Here we discuss the main DFS concerns that are independent of a single DFS implementation.

## 2.1 Transparency

DFS clients, like any file system client, should be unaware of the storage mechanisms. This means that the number of servers and the "dispersion of storage devices" should be transparent to a client of the DFS [7]. The main concept of transparency is network transparency. A client should be able to "access remote files using the same set of file operations applicable to local files [7]." This places the responsibility of locating files and transmitting data across the network on the DFS.

## 2.2 Fault Tolerance

A DFS should be able to perform in the case of faults. Levy and Silberschatz use a broad definition of a "fault" which includes communication faults, machine failures, storage device crashes, and the decay of storage media. In the case of these faults, the DFS needs to be able to function, but the level of continued system functionality can vary. It is often acceptable for performance to degraded in proportion with the severity of the fault [7].

### 2.2.1 Availability

*Availability* is an additional criteria that can be imposed on a DFS with respect to fault tolerance. A file is "available" if it can be accessed regardless of storage failures and machine faults [7]. A similar criterion is *robustness*, which means that the file is guaranteed to survive faults, but it may not become available until faults are recovered [7].

### 2.2.2 Replication

In order to increase the availability of files within a DFS, DFS designers can use replication strategies. One common mechanism for replication is "demand replication" where files have a primary replica, which is accessed first for read and write requests, and supplementary secondary replica(s) which can be used for access during faults on the primary machine [18].

## 2.3 Scalability

*Scalability* is "the capability of the system to adapt to increased service load [7]." When compared to a traditional file system, a DFS has a higher risk of saturation because communication overhead is associated with DFS client requests since the DFS must send and receive data over a network. However, a distributed system can gain benefits from having multiplicity of resources [7]. Centralized components of any distributed system remove the benefits of resource multiplicity and introduce a bottleneck on the resource, which can be the cause of system faults.

Scalability and fault tolerance are related since the act of scaling can cause faults, and some faults may hinder the system's ability to properly scale. In order to build a system that is both scalable and fault tolerant, a DFS must have multiple, indepen-

dent servers that control multiple, independent storage devices [7].

## 2.4 File Names

All file systems have a layer of abstraction between a textual file name, and the actual storage of the data in disk blocks. A DFS introduces another layer of abstraction for transparency as described in Section 2.1. To support transparency, a DFS must hide where in the network of resources file data is located. A DFS may also hide this information with file replicas (2.2.2), multiple copies of file data to support fault tolerance (Section 2.2). With all abstractions, a DFS must maintain a mapping of a filename to a list of all storage locations for all pieces of a DFS file, hiding the existence of replicas and storage locations [7].

### 2.4.1 Additional Naming Properties

Two new properties are imposed on a DFS with regard to transparency through naming [7]. In order to ensure that DFS Clients are unaware of storage mechanisms (Section 2.1), these naming properties must be enforced by a DFS.

1. Location Transparency: Filenames do not expose storage location information.
2. Location Independence: Filenames do not need to be changed when physical storage location changes.

# CHAPTER 3

# Related Work

Both research and industry use Distributed File Systems, but to the best of our knowledge, no extant system can be broken up into pieces conducive to student learning.

## 3.1 Andrew File System (AFS)

Andrew File System (AFS) is a distributed network file system [16]. AFS is unique to DFS research because it provides "location independence, scalability and transparent migration", while working across Operating Systems. AFS files are organized into a "globally unique namespace" as seen in Figure 3.1 [22]. AFS facilitates these non-server identifiable namespaces by maintaining a replicated location database. Clients must connect to the database to resolve file names and find data. These domains are considered to be "AFS cells" and file pathnames include cells rather than server names [22]. This namespace design allows for location transparency and independence as described in 2.4.1, since AFS administrators can move data between servers without affecting clients. This model also addresses scalability (Section 2.3) since more resources can be added in a particular domain without notifying the clients.

With all of these features, "the different aspects of AFS can be overwhelming at first and the learning curve for setting up your own AFS cell is steep [22]." AFS justifies the steep learning curve because "secure, platform-independent world-wide file sharing is a concept as attractive as serving your `/usr/local/` area and all your UNIX home directories [22]." AFS is designed for secure global sharing, and is main-

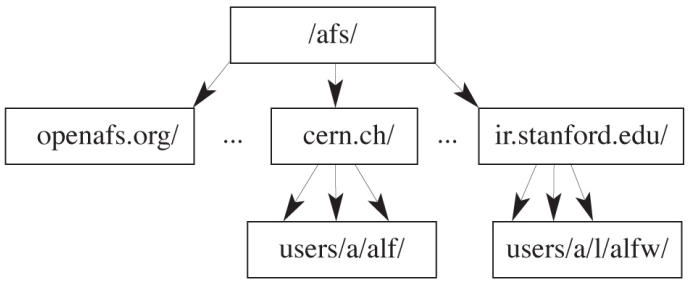


Figure 3.1: Shows the namespace design for AFS files [22].

tained today as an open source project [14]. However, these design considerations, and the learning curve, make the system too complex for small, meaningful, DFS behavioral modifications in the classroom.

## 3.2 Google File System (GFS)

The Google File System (GFS) is a "scalable distributed file system for large distributed data-intensive applications [5]." GFS is designed with similar goals discussed in CHAPTER 2, such as scalability (2.3) and availability (2.2), but is tailored to Google's application needs [5]. For example, Google constantly deals with component failures (2.2) due to the "quantity and quality of the components" which guarantees that "some are not functional at any given time and some will not recover from their current failures [5]." For Google's use, most file modifications are append only, and once written to, "files are only read, and often sequentially [5]."

In addition to GFS being proprietary, some of the design considerations made for GFS make it unusable for students. Such considerations are as follows:

- "We expect a few million files, each typically 100 MB or larger in size. Multi-GB files are the common case and should be managed efficiently. Small files must be supported, but we need not optimize for them [5]."

At Cal Poly we support "Learn By Doing" and encourage students to have hands on experience in as many aspects of a project as possible. Therefore, it is desirable for students to collect and use their own data for projects, making smaller files more common than large files due to the time required for data collection.

- "The workloads primarily consist of two kinds of reads: large streaming reads and small random reads...The workloads also have many large, sequential writes that append data to files [5]."

  The goal of DecaFS is flexibility, students should be able to modify the system to adapt to their needs for a given project. Therefore, we cannot predict the workloads since they vary from project to project and by student.

- "The system must efficiently implement well-defined semantics for multiple clients that concurrently append to the same file [5]."

  Cal Poly course sizes are small, and we will have significantly less hardware running our DFS than at Google, so we are able to impose write restrictions that allow us to have only one client writing a file at a time.

### 3.2.1 GFS Architecture

Even though a system like GFS is infeasible for our purposes, many other systems are based on GFS, so it is still meaningful to explore the GFS architecture [5] as seen in Figure 3.2. A GFS cluster has a single master node, and multiple chunk servers (worker nodes) and can be accessed by multiple clients. Every file is divided into chunks of a fixed size, identifiable by a unique "chunk handle" that is assigned by the master node on creation. Chunks are stored on disk in chunk servers, and each chunk is replicated three times by default. The master node is responsible for

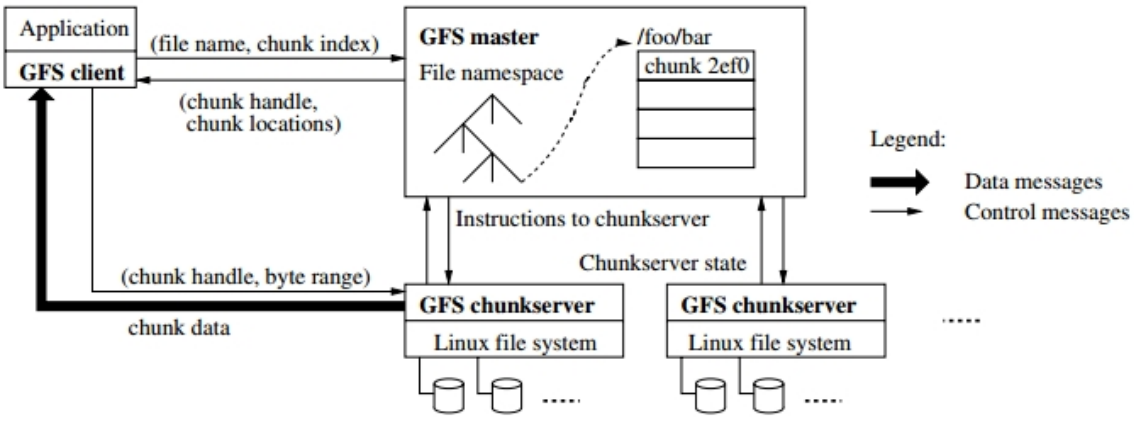


Figure 3.2: Architecture for GFS [5].

the maintenance of metadata and system-wide activities. The master communicates regularly with chunk servers via heartbeats that are periodic checks for state.

## 3.3 Hadoop Distributed File System (HDFS)

Hadoop Distributed File System (HDFS) is an open-source project similar to GFS. The HDFS architecture can be seen in Figure 3.3. Assumptions for HDFS, like GFS, include hardware failures, streaming data processing (batch processing), large data files, and "write-once-read-many" file access [15]. HDFS has a master-slave architecture, with one master (NameNode) and many slaves (DataNode) [15]. Files are stored as "blocks" and each file is split into one or more blocks (like GFS chunks) [15].

In addition to the requirement differences discussed in 3.2 which also apply to HDFS, HDFS cannot be directly mounted in user space, which limits the usefulness for students. Like GFS, we have determined that HDFS is unsuitable for classroom learning. The HDFS code-base is very large, and HDFS has many features for large-scale data processing that are not useful for the amount of data students can easily

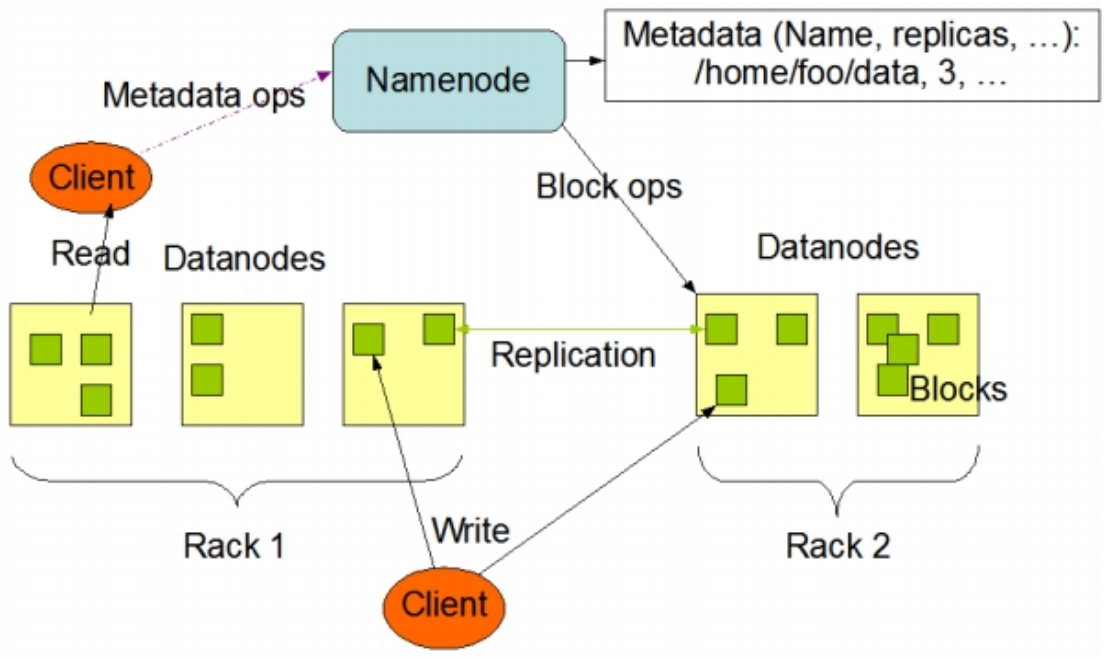


Figure 3.3: Architecture for HDFS [15].

collect.

## 3.4 Others

Other systems have been implemented that are similar to GFS and HDFS. For example, Quantcast File System (QFS) is a C++ HDFS alternative based on Kosmos Distributed File System (KFS) [1], implemented to work with the HDFS APIs. KFS, QFS, and other systems like it, are performance improvements over HDFS. These and other systems [10, 17] are implemented with similar design considerations of HDFS and GFS [9].

## 3.5 Related Work and DecaFS

We have explored a sample of Distributed File Systems that are available, and found that, in general, our design requirements discussed in CHAPTER 2 are accurate. Extant systems focus on scalability, fault tolerance, and transparency for real data sets. These system impose stricter requirements and more complex implementations than are needed for a classroom setting. Since extant systems are too robust or complex for student use, we have decided to build DecaFS for educational purposes. DecaFS is heavily inspired by DFS systems covered in this chapter, but simplified down to the most basic DFS components and functions to reduce the learning curve for students and make biweekly projects feasible.

# CHAPTER 4

# DecaFS Requirements

## 4.1 System Requirements

Cal Poly courses need projects that allow students to develop pieces of distributed systems to learn different distribution and distributed system management techniques. They must also be able to build applications that use a distributed system to solve problems from various application domains. The high-level requirements of the system are seen as follows:

- REQ-1: Students shall be able to develop architectural components of a distributed system
- REQ-2: Students shall be able to build applications for a distributed system

We have decided to develop a modular distributed file system (DecaFS) that allows students to develop components (modules) of the DFS. Altering modules serves two purposes: first, students can develop components of a distributed system, the DFS, (REQ-1), second, students can alter the behavior of the DFS to support their data needs for applications (REQ-2).

## 4.2 DFS Requirements

In addition to the overall system requirements described in Section 4.1, several requirements have been placed on the DFS to ensure that the system can be adapted

to the needs of various projects and courses.

- REQ-3: Students shall be able to change which node(s) data is stored on/recovered from.
- REQ-4: Students shall be able to change the replication policies of the system. The system should support no replication, mirroring, and some RAID [3] implementations.
- REQ-5: The DFS should be mountable with FUSE [21].
- REQ-6: The system shall be able to tolerate at least one worker-node failure.

### 4.3 Limitations and Configuration

The DFS also needs to have limitations and configuration properties that can change per DFS-instance.

- REQ-7: DecaFS System Administrators shall be able to set the maximum possible file size for the DFS.
- REQ-8: DecaFS System Administrators shall be able to set the size (in bytes) of the stripes for each file, where a stripe is the maximum number of bytes of file data that are broken up into pieces and distributed for storage.
- REQ-9: DecaFS System Administrators shall be able to set the size (in bytes) of the chunks for each file, where a chunk is the maximum number of bytes of a stripe of file data stored at a time by one write to one storage node.

# CHAPTER 5

# DecaFS Design

We designed DecaFS, Distributed Educational Component Adaptable File System, to meet the requirements specified in CHAPTER 4, to provide students with a flexible DFS that can be customized to include an application need or distributed systems concept.

## 5.1 Overview

We designed DecaFS to have a master-slave architecture similar to GFS 3.2 and HDFS 3.3. Our design is similar to these two systems, but scaled down for smaller numbers of nodes in the DFS and fewer clients. Our requirements are therefore more relaxed than those of GFS and HDFS. We designed to support one-node failures, smaller file sizes, and did not attempt to optimize our system for performance.

DecaFS is broken down into three functional layers. The Barista Layer (5.3) contains all functionality of the master node of the system and the Espresso Layer (5.4) contains all worker functionality. Communication between Barista and Espresso is also separated in its own layer, the Network Layer (5.5). Each layer is broken up into modules, which are logical components of the DFS. Each module is responsible for one task, or a set of tasks, that accomplish a piece of DFS functionality.

Students should be able to replace a module, a set of modules, or one or more layers of DecaFS by following the module or layer APIs to achieve desired DFS behavior. With this modular design, students can change the behavior of the system, develop

pieces of the system that are not provided by a professor, or improve extant system modules to learn about distributed systems.

The overall architecture for DecaFS can be seen in Figure 5.1, and each module will be explained in more detail in the rest of this chapter, and in CHAPTER 7. DecaFS modules work together to accomplish DFS tasks which are a subset of Unix File System functionality. DecaFS supports the following operations:

- File: open, read, write, close, remove, seek, stat
- Directory: make, open, read, remove

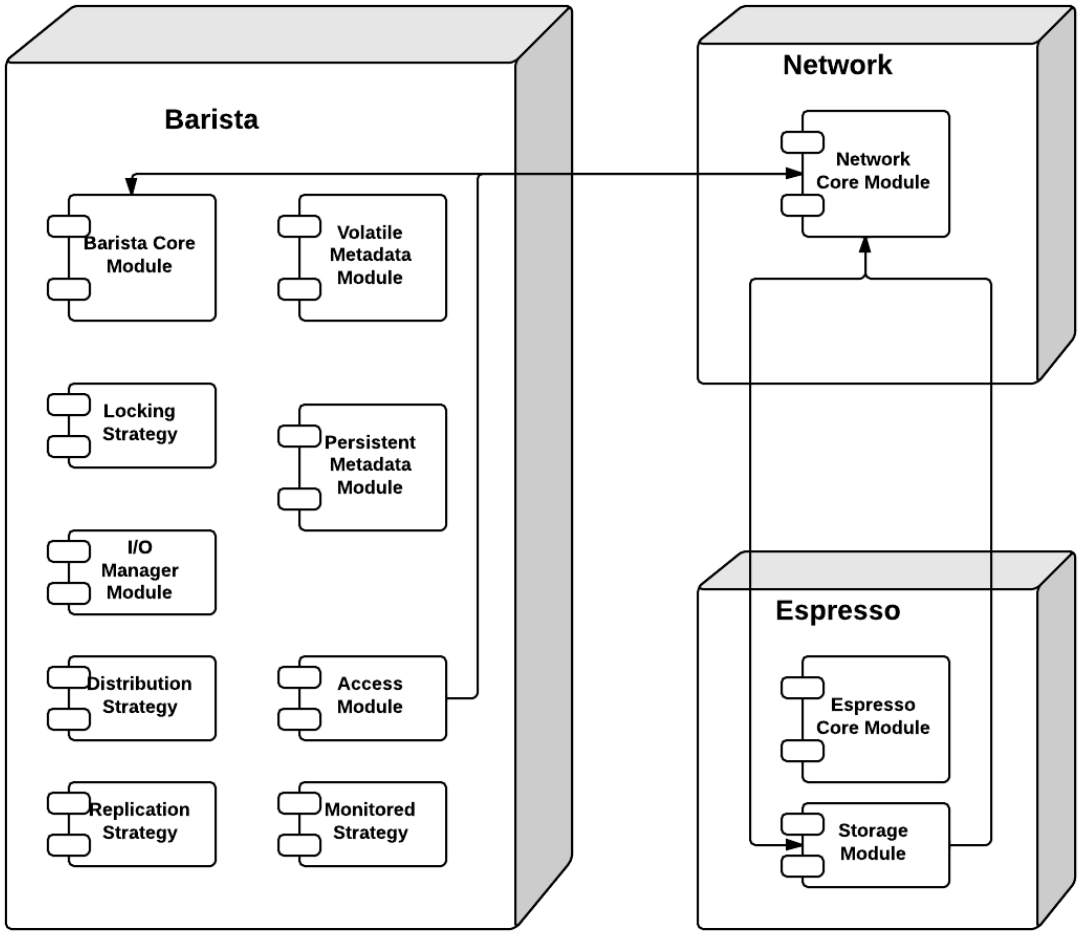


Figure 5.1: Architecture for DecaFS.

## 5.2 Definitions

In order to properly discuss the functions of DecaFS Modules and the interactions between them, we need to define terms that we use throughout our system.

- *Node*: a machine running a component of DecaFS.
- *Barista:* the master node, one per instance of DecaFS.
- *Espresso:* a worker node, one or more per instance of DecaFS.
- *DecaFS Client:* any user running a process on a node in DecaFS. One user running multiple processes on the same node is considered the same client. One user running multiple processes on two different nodes are considered two separate clients.
- *Stripe:* a logical piece of a file to be distributed across multiple Espresso nodes in DecaFS.
- *Chunk:* a piece of a Stripe, the actual data sent to or read from an Espresso node.

## 5.3 Barista

As described in Section 5.1, the Barista Layer contains all functionality for the master node of DecaFS. Barista Responsibilities include: metadata management, monitoring system health (node failures), processing DecaFS client requests, and managing synchronization.

### 5.3.1 Barista Core

Barista Core is the center of the Barista Layer. The goal of this module is to control the flow of requests throughout the system, and should not be modified by students in most cases. Barista Core is the main entry point for the Barista Node from DecaFS Clients. While Barista Core manages the flow of control of the system, the functionality of the system lies in other modules. This allows students to change the behavior of the system, without risk of broken communication channels between nodes.

Barista Core handles communication with the Network Layer, so that network information can be hidden from students. This removes the need of explicit knowledge of networks to develop a module for DecaFS. Barista Core also manages the storage and recovery of System Metadata. This ensures that DecaFS can clean-up and start-up without relying on students to log when they are reading and writing data from other modules.

Barista Core deals with data of various sizes from the DecaFS Clients, and breaks requests down into stripes before sending requests to other Barista Modules. In addition to handling DecaFS Client requests, Barista Core implements a C-API for students to use to gather System Information from any module. The C-APIs are discussed further in CHAPTER 7.

### 5.3.2 Persistent Metadata

Barista Core uses Persistent Metadata to store all system information. This information includes the list of files stored in DecaFS and general information for each file such as size, id, and the stripe and chunk size associated with that file.

### 5.3.3 Volatile Metadata

Barista Core uses Volatile Metadata to maintain the state of the current instance of DecaFS and the interaction between the system and its clients. Volatile Metadata is in charge of maintaining information such as the list of nodes currently "active", or not in a failure state, and the set of file descriptors for DecaFS clients.

### 5.3.4 Locking Strategy

Barista Core uses the Locking Strategy module to manage DecaFS Client accesses to files at any point in time. For simplicity, DecaFS allows only one DecaFS Client to access a file at a time, but this can be changed by modifying or re-writing the Locking Strategy module.

DecaFS maintains two types of locks: exclusive and shared. Exclusive locks are needed to write to a file, and shared locks allow reading from a file. By default, only one process under one client may have an exclusive lock on a file at any point in time. Many processes under the same client may hold a shared lock on a file at any point in time. Locks are assigned based on the permissions that a file is opened with, and released when the file is closed.

We do not permit upgrading or downgrading of locks, so in order to switch from a shared lock to an exclusive lock or vice versa, DecaFS Clients must close and re-open the file. This allows DecaFS to avoid deadlocks and alleviate lock starvation.

### 5.3.5 IO Manager

IO Manager is responsible for breaking down tasks (read/write/delete) and distributing work among the Espresso Nodes of the system. Students can modify this module to customize the mechanisms for storage and retrieval of data in/from Espresso Nodes.

IO Manager receives requests of stripe size from Barista Core and is responsible for breaking striped requests into chunks for the Espresso Nodes. This allows students to customize behavior since they have total control over the size and number of chunks created per stripe. This module is also responsible for managing data replication policies.

### 5.3.6 Distribution Strategy

The Distribution Strategy Module provides the mechanism for determining what node a particular chunk should be sent to. A chunk is uniquely identified by its file, stripe and chunk number. This module allows students to change where chunks are sent throughout the system without writing an entire IO manager as described in 5.3.5.

### 5.3.7 Replication Strategy

Similar to the Distribution Strategy Module (5.3.6), the Replication Strategy Module provides a mechanism for determining to which node a particular chunk's replica should be sent. Again, students can alter where chunk replicas are sent without writing an entire IO Manager Module (5.3.5).

### 5.3.8 IO, Distribution and Replication

The combination of these three modules gives students total control over the mechanisms for distributing file chunks and replicating data. With this control, students can choose to not replicate their data, to do mirrored replication where each chunk is written to two nodes, or to implement more complex strategies such as RAID, without managing system metadata.

If students wish to implement more complex storage and replication strategies

that require additional metadata, we allow them to register their own functions to be called on system start-up that will allow them to recover any additional metadata they wish to store.

### 5.3.9 Access Module

The access module is the end point for data processing on the Barista Layer. It receives read, write, and delete requests from other Modules and sends them to the Network Layer. The Access Module is a good place for students to add a buffering system to avoid extraneous network calls.

### 5.3.10 Monitored Strategy

Students may also wish to perform system monitoring for tasks such as re-balancing data if their storage mechanisms are not balanced. We allow them to register functions with Barista Core for monitoring that will be called incrementally based on a time-out specified at the time of registration.

Two monitoring functions are pre-defined for students and will be called each time the system detects a node failure, or node coming back online in case a student's module needs to be notified.

## 5.4 Espresso

As described in Section 5.1, the Espresso Layer is responsible for storing data on disk and maintaining all associated metadata. Students should not have to alter any component of the Espresso Layer. The layer facilitates raw-data storage, but all logic related to the storage of file data resides in the Barista Layer.

### 5.4.1 Espresso Storage

Espresso Storage is the main module of the Espresso Layer. It is responsible for executing chunk-level read, write, and delete requests. This module manages local file data and associated metadata and knows only about which chunks exist on the Espresso Node the module runs on.

## 5.5 Network

As described in Section 5.1, the Network Layer handles communication between nodes in DecaFS and between clients and the system. This layer hides the notion of packets from the rest of the system, allowing students to develop DecaFS modules without prior knowledge of networks. Students will need to understand that nodes in the system communicate with one another, but will use a DecaFS API to handle communication, rather than using direct network calls and buffers.

### 5.5.1 Net TCP

The Net TCP module provides functionality that is common to multiple network components of DecaFS. This functionality includes defining server behavior, allowing a node to listen for requests, defining client behavior, sending requests to a server, and management of the connections between clients and servers.

### 5.5.2 Network Core

The Network Core module extends the functionality of the Net TCP module to be specific to the needs of different nodes in DecaFS. This module defines the different packet types required for communication between Barista and Espresso nodes, as well as DecaFS Clients. It also handles the processing of packets and exposes an API for

students to use to send requests across the network.

## 5.6 Connecting Layers

### 5.6.1 DecaFS Barista

The DecaFS Barista Module is responsible for set-up of the Barista Node. It ensures that all metadata is recovered through Barista Core, and connects itself to the system through the Network Layer.

### 5.6.2 Espresso Core

The Espresso Core Module is responsible for set-up of an Espresso Node. This layer must restore any metadata, and connect itself to the system through the Network Layer.

# CHAPTER 6

# DecaFS Workflows

DecaFS is designed for students to write modules, so the functionality of the DFS has to be broken down into pieces small enough for student to easily work with. However, this design makes it difficult to trace the flow of execution throughout the system for different tasks. In this section, we examine workflows of major DFS tasks, and follow the flow of execution through DecaFS modules.

For simplicity in all discussion of workflows, we will ignore the Network Layer. Please assume that all communication between Barista and Espresso Modules, and between the DecaFS Client and the Barista Core goes through the Network Layer.

## 6.1 Data Flow

We have briefly discussed data flow throughout DecaFS in terms of stripes and chunks. DecaFS modules are responsible for dealing with different data fragments. Each module that deals directly with data, is designed to fragment the data only once. The overall data flow for the system can be seen in Figure 6.1. DecaFS Clients request or send raw data to DecaFS of any size, within DecaFS file size limits. Raw Data goes directly to Barista Core, the entry-point for the Barista Node (5.3). Barista Core then breaks up the request into stripe-size requests. Striped data is sent/requested from the IO Manager, which further breaks down the request into chunk-sized requests for per-node storage or retrieval. Chunks are then sent to/requested from Espresso Nodes through the Access Module. The Access Module does not transform the data

further, but translates the Barista's request into a Network Request to be sent to an Espresso Node. Once a request is received on the Espresso Side, all data is dealt with at the chunk level.

Responses or data requested is sent from the Espresso Layer back to Barista Core, since Barista Core maintains information about the entire request from the DecaFS Client. Once all fragments of the request have been received, Barista Core sends a response to the DecaFS Client via the Network Layer.

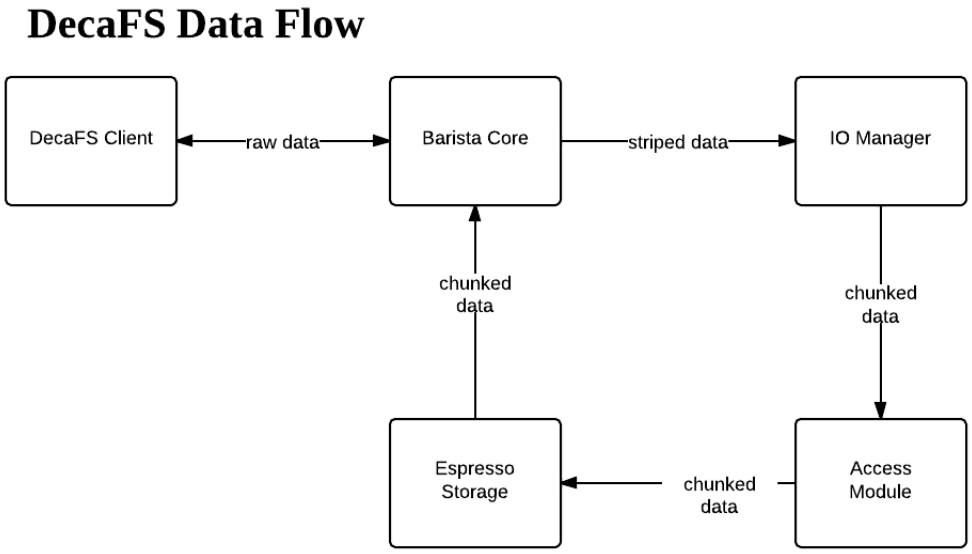


Figure 6.1: The process of data fragmentation through DecaFS.

## 6.2 Open

Opening a file in DecaFS happens only in the Barista Layer. We allow files to be opened either as read only or read/write, with the file cursor starting at 0 or end of file. Files opened for read/write are automatically created if they do not exist.

The normal flow of execution for open can been seen in Figure 6.3 and Figure 6.4,

and failures can be seen in A.1. If a DecaFS Client requests to open a file, Barista Core first checks if the file exists through `decafs_file_stat` in the Persistent Metadata Module. If the file does not exist based on the file stat call, and the DecaFS Client wants to write to the file, it is created at this time. If the DecaFS Client attempts to open a non-existent file for reading, the open call will return an error. Then, Barista Core uses the Locking Strategy Module to obtain a lock (shared for read only, exclusive for read/write) on the file. Finally, Barista Core gets a file descriptor from the Volatile Metadata Module, which maintains associated metadata for that file descriptor throughout the time that the new instance of the file is open.

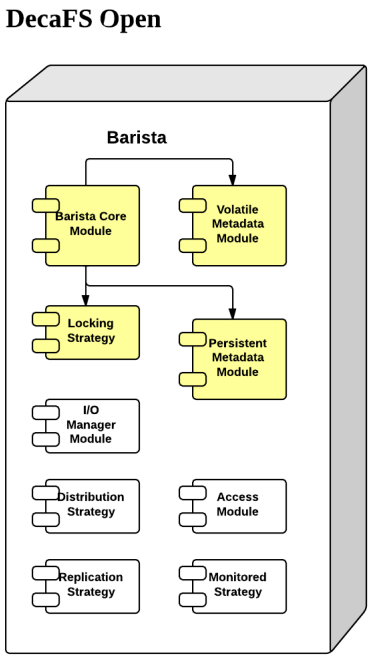


Figure 6.2: Components involved with a DecaFS open call.

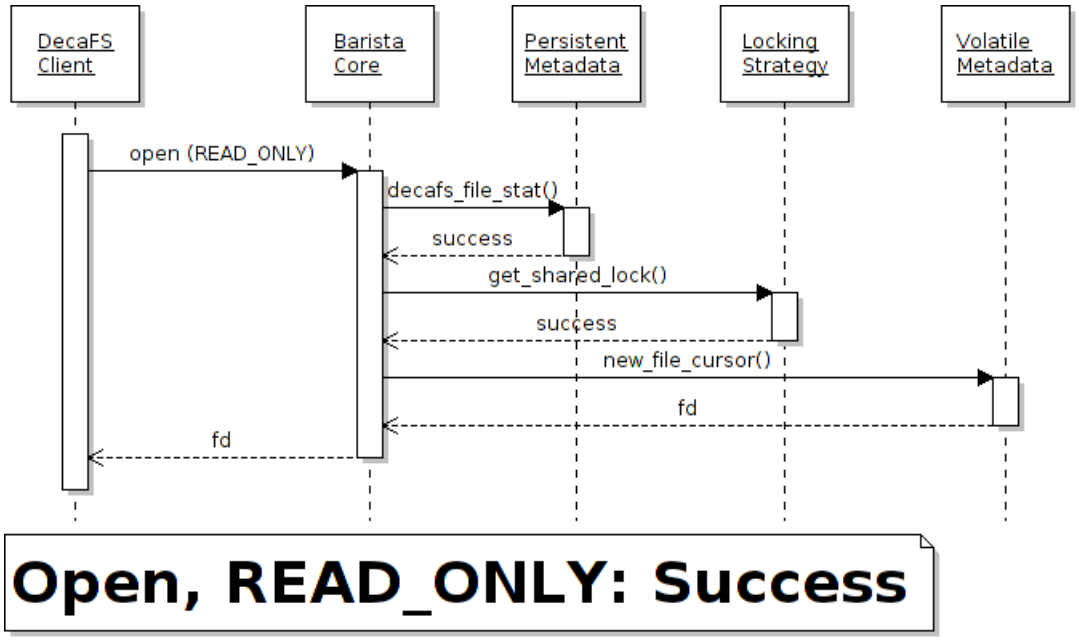


Figure 6.3: A successful call to open for reading a file.

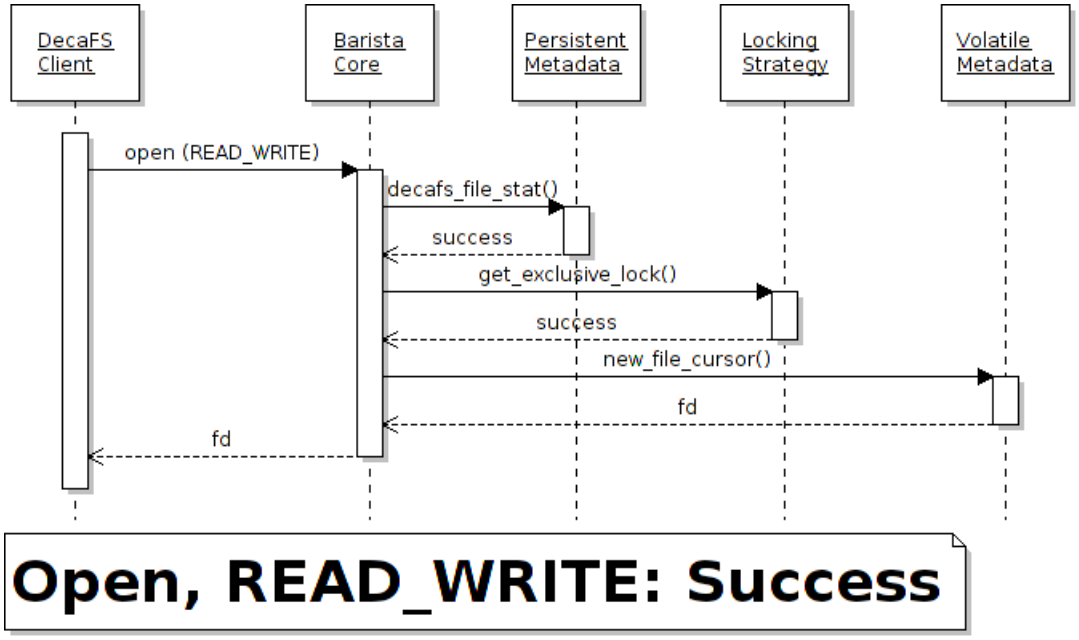


Figure 6.4: A successful call to open for reading/writing a file.

## 6.3 Read

Reading a file in DecaFS goes through every layer of the system, and involves most of the modules in the Barista Layer. A normal execution for read can be seen in Figure 6.6. When a read request comes in from a DecaFS Client, Barista Core will automatically break the read request into stripes. Stripes are then forwarded to the IO Manager Module, which is a component that students should be able to implement. Before processing moves to IO Manager, Barista Core will ensure that the file exists, a read or a write lock is held for the caller, and that the file is open for reading or both reading and writing.

The IO Manager Module is responsible for breaking the stripe into chunks, determining what node a chunk is stored on, and determining whether the chunk needs to be read from the regular storage or a replica storage (if available). It is then the responsibility of the IO Manager to check the "health" of the system, and only send requests to nodes that are "up", meaning the nodes that have not received a failure. The API to determine system health is exposed in the Volatile Metadata Module. The IO Manager is also responsible for handling node failures, more information can be found in Section 7.6.

IO Managers must break striped read requests into chunk-sized requests and send them to the Access Module, which will then handle the translation to the Network Layer. For each stripe processed, the IO Manager must inform Barista Core how many chunks were used in breaking down the striped request. Finally, the chunk request arrives on the corresponding Espresso Node, which returns the data.

Read responses from the Espresso Layer contain the different chunks requested by the IO Manager. These responses are sent to Barista Core, which waits for all chunks requested to arrive. Once the module has received all chunks, it assembles the full

read buffer and sends the final read response to the DecaFS Client.

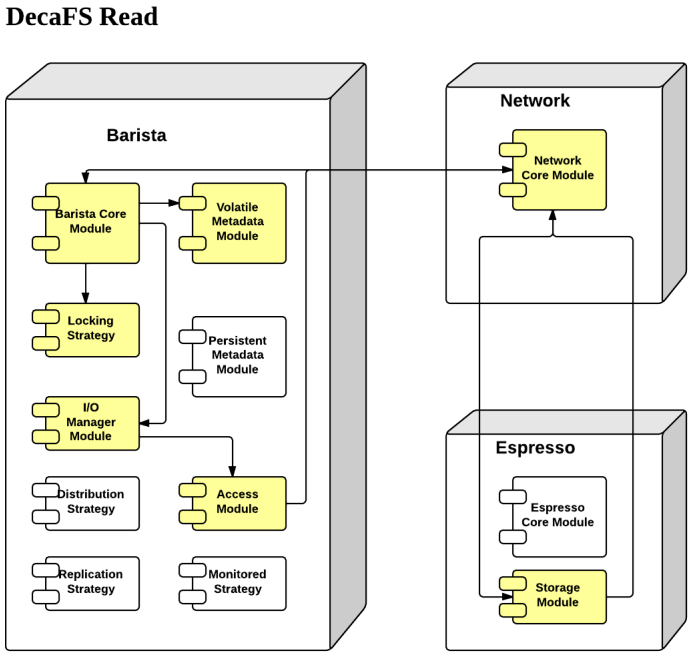


Figure 6.5: Components involved with a DecaFS read call.

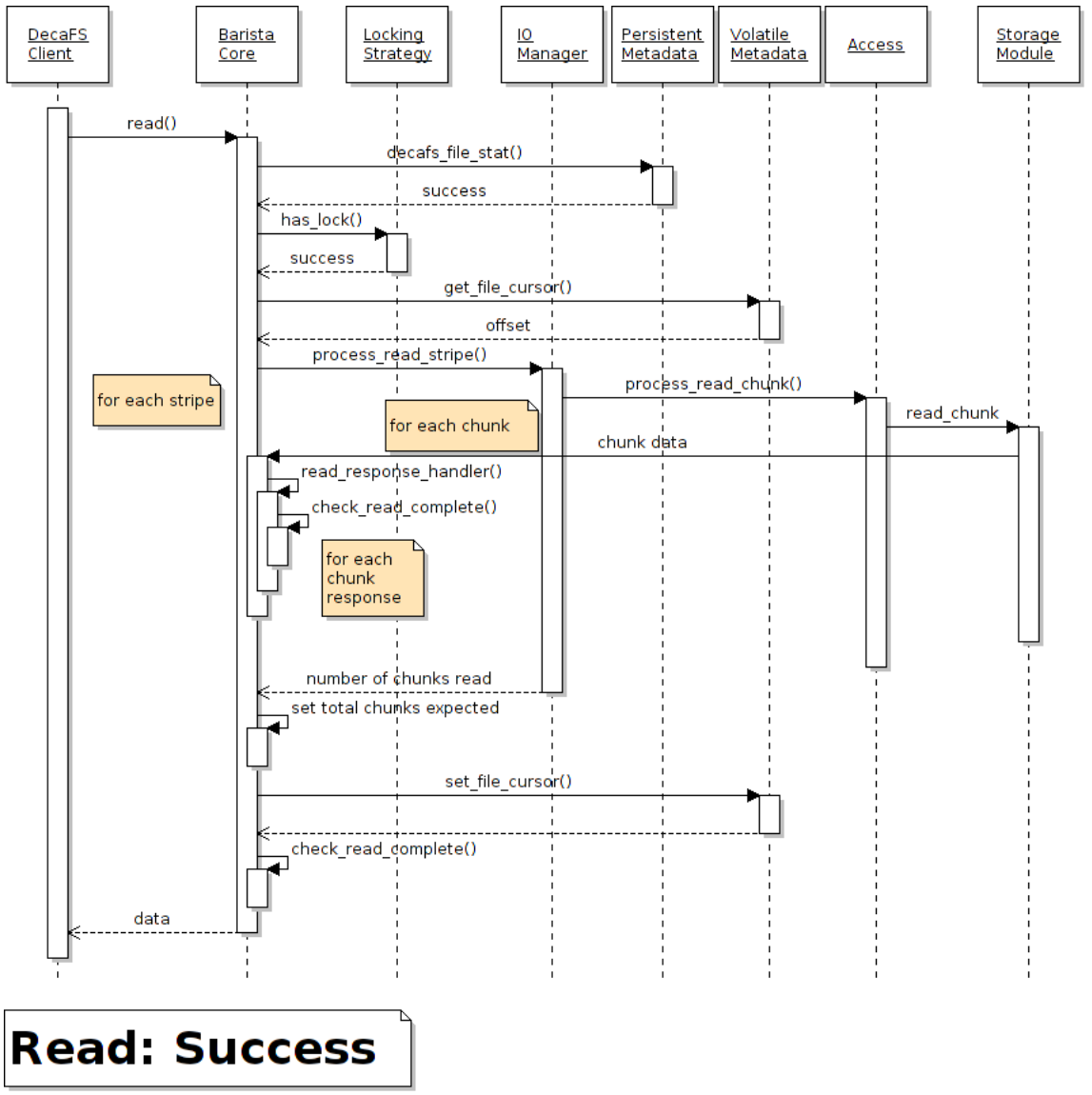


Figure 6.6: A successful call to read.

## 6.4 Write

Like Read (Section 6.3), writing to a file goes through all layers and most modules of DecaFS. A non-erroneous execution of a write call can be seen in Figure 6.8. Similar to a read request, when a write request arrives in the Barista Core Module from a DecaFS Client, Barista Core will break up the write request into stripe-size requests.

Before processing moves to IO Manager, Barista Core will ensure that the file exists, a write lock is held for the caller, and that the file is open for write. Stripes are forwarded to the IO Manager Module for chunk-level processing.

Once the striped-request reaches the IO Manager, it is the IO Manager's responsibility to break down the request into chunk-sized requests and send these requests to the Access Module, which translates them into Network Requests. Chunk write requests arrive on the Espresso Layer in the Storage Module, where data is written to disk. Chunk Write Responses are sent back to Barista Core. As in the case of a read request, Barista core waits for all write responses to arrive before notifying the DecaFS Client that the write is complete. Barista Core maintains global metadata affected by the write such as file size and moving the file cursor.

### 6.4.1 Metadata

The IO Manager is responsible for maintaining metadata for determining what node each chunk is stored on. For each stripe processed, the IO Manager should update its local metadata when necessary. The IO Manager is later responsible for utilizing local metadata to locate previously-written chunks for read requests.

### 6.4.2 Chunks and Replica Chunks

It is also the responsibility of the IO Manager to facilitate Replication to ensure that DecaFS is Fault Tolerant. While processing a striped write request, there is no limitation on the number of chunk write requests sent or replication requests. The only limitation we impose on the IO Manager is that for each stripe, the IO Manager must utilize return parameters to notify Barista Core of the number of Chunk and Replica requests generated in the processing of the stripe. This allows Barista Core to handle Network Write Responses and notify the DecaFS Client on completion of

the full write request.

### 6.4.3 Distribution and Replication Strategy Modules

We provide a default IO Manager to the students that utilizes Mirrored Replication. Our IO Manager uses two sub-modules: Distribution Strategy and Replication Strategy to determine the node that each chunk (or replica) should be sent to. This allows students to alter the storage location of chunks (and replicas) without writing an entire IO Manager. More information about our Mirrored IO Manager can be found in Section 7.6.

### 6.4.4 Writes During Node Failures

Writing to new files (or new chunks) can work as described above, if the IO Manager is aware of node failures. For new files, chunks and replica data would be assigned only to nodes that were up at the time of the write. However, updating extant chunks is an issue if a node involved in the chunk/replica write is down. The issue can be described as follows: we have data written on node one with replica data on node three. If node one goes down, the replica can be updated. However, if node one comes back online, or is active again on system start-up at a later date, node one contains stale data.

This issue, and other similar issues, is the motivation behind the Monitored Strategy Module (5.3.10). Monitored Strategy contains mechanisms that enable background processes to do system clean-up, such as this write issue Section 7.11. In order to resolve stale data from writes during node failures, our recommendation is as follows:

1. If a write request is to extant chunks, write only to nodes that are currently up

2. Maintain a set of work to be completed in the background.

   We recommend use of the Persistent Set (Section 7.2). This data needs to persist in case failed nodes do not come online through the execution of DecaFS since these failures will need to be resolved before a later execution of DecaFS can proceed without error.

3. When a node comes online (we provide a callback for this in Monitored Strategy Section 7.11), check the set of work for background tasks that the newly "online" node needs to complete.

4. For each task, send write requests to the Espresso Node via the Access Module (Section 7.10) and notify Barista Core of the request (Section 7.1).

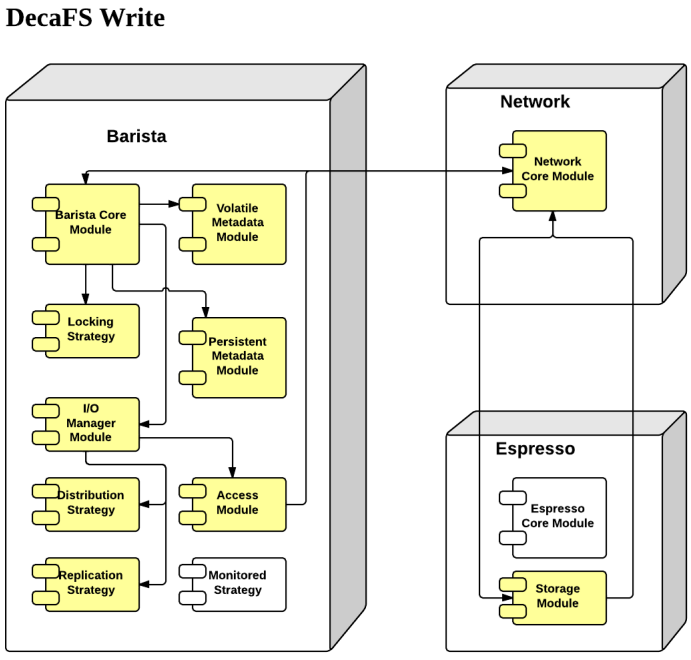


Figure 6.7: Components involved with a DecaFS write call.

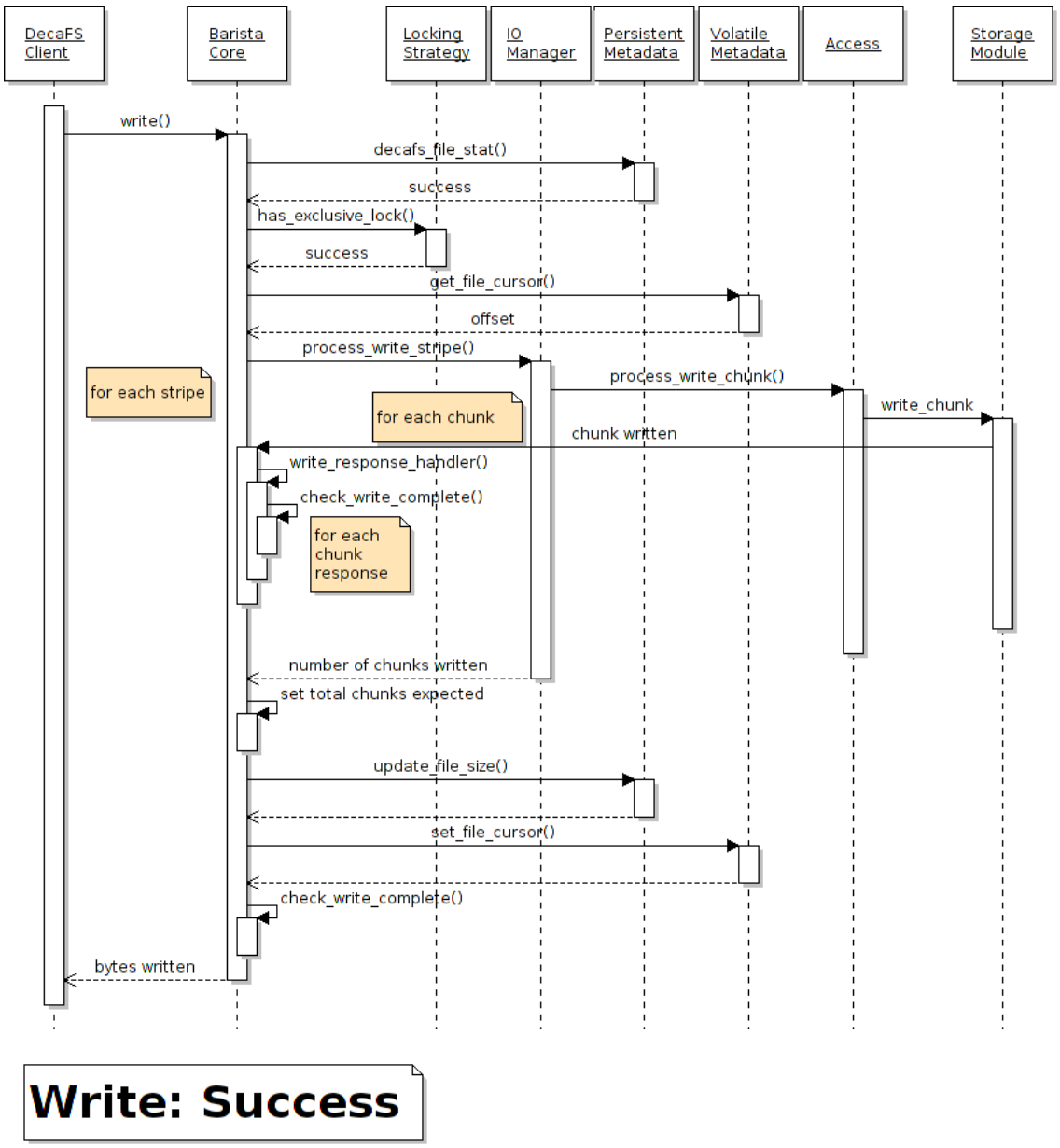


Figure 6.8: A successful call to write.

## 6.5 Close

Similar to Open (Section 6.2), the close call does not leave the Barista Layer. When a DecaFS Client requests to close a file, Barista Core uses Volatile Metadata to remove the file cursor representing the open instance of the file. If the cursor does not exist,

or if the calling DecaFS Client is not the owner of the file cursor, this call will fail as seen in A.4. If removal of the cursor is successful, the lock held on the file will be released. Components involved with close can be seen in Figure 6.9 and a successful call to close can be seen in Figure 6.10.

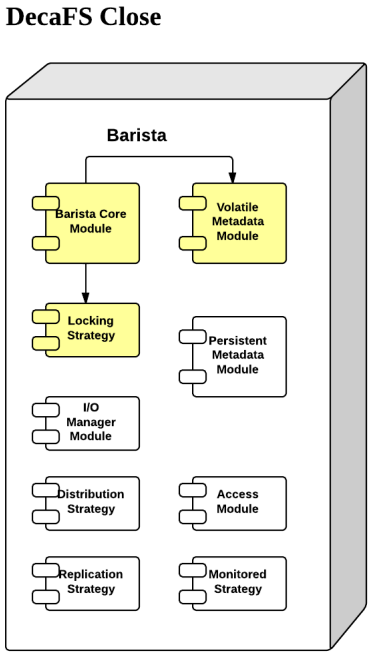


Figure 6.9: Components involved with a DecaFS close call.

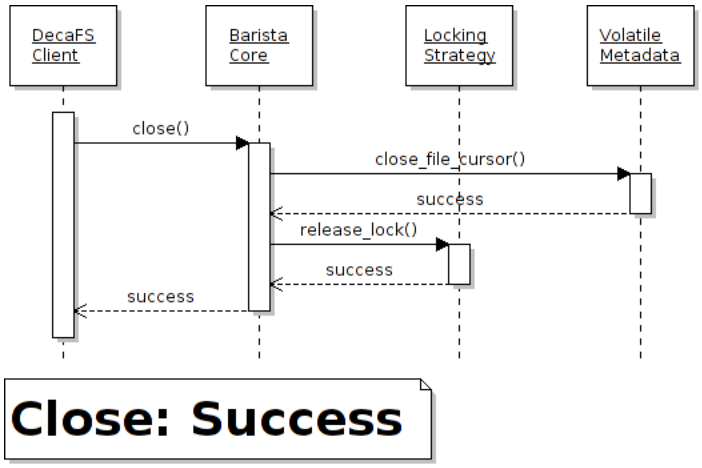


Figure 6.10: A successful call to close.

## 6.6 Delete

Similar to Read (6.2) and Write (6.4), the process of deleting a file utilizes every layer of DecaFS. When Barista Core receives a delete request from a DecaFS Client, it ensures that the file exists, and obtains an exclusive lock on the file from the Locking Strategy Module. Errors in either call result in a failed delete as seen in A.5. Barista Core then sends the delete request to the IO Manager.

The IO Manager must determine all of the chunks that correspond to the file (6.6.1) and send a request to the Access Module to delete each chunk. The IO Manager must also notify Barista Core of the number of chunks it requested to delete. This number includes both chunks and potential replica chunks.

Requests pass through the Access Module and are translated into Network Requests. Chunk level delete requests arrive on the Espresso Layer in the Espresso Storage Module and are deleted from disk. Espresso Storage Responses go through the Network Layer automatically and arrive at Barista Core. Barista Core waits for all chunk responses to arrive before responding to the DecaFS Client that the delete has completed.

A successful call to delete can be seen in Figure 6.12 and components involved can be seen in Figure 6.11.

### 6.6.1 Metadata

IO Manager must utilize its local metadata discussed in Section 6.4 to locate all chunks (and replica data) for a file on a delete call. This responsibility is due to the fact that deletion is a chunk-level operation, and the IO Manager is responsible for chunk-level processing, as discussed in Section 6.1.

### 6.6.2 Node Failures

It is also the responsibility of the IO Manager to determine how to handle node failures during or before a delete call. We recommend that the IO Manager skips delete chunk calls to nodes that are down at the time of the call. Students can later clean up missed chunks using a Monitored Strategy discussed in Section 7.11. Without skipping these chunks, the IO Manager would either have to fail the delete call completely, wait for all nodes to come back up before processing the delete, or leave chunks on nodes that are down indefinitely.

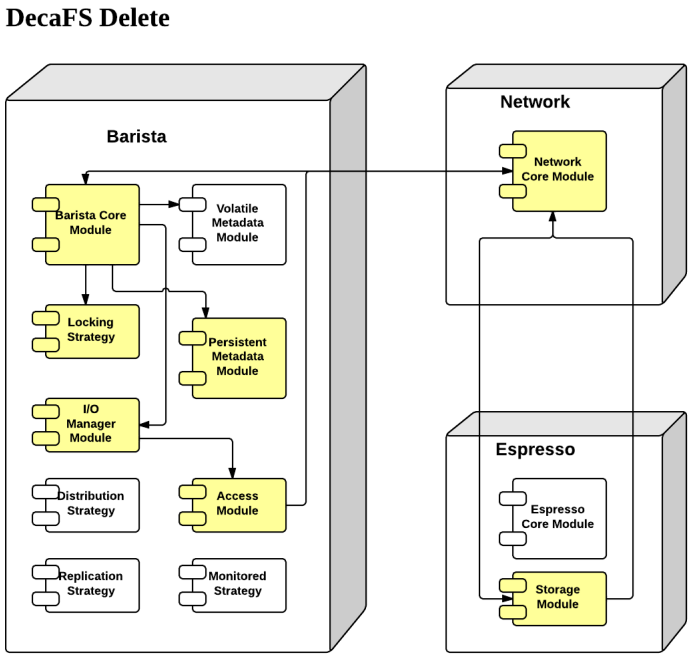


Figure 6.11: Components involved with a DecaFS delete call.

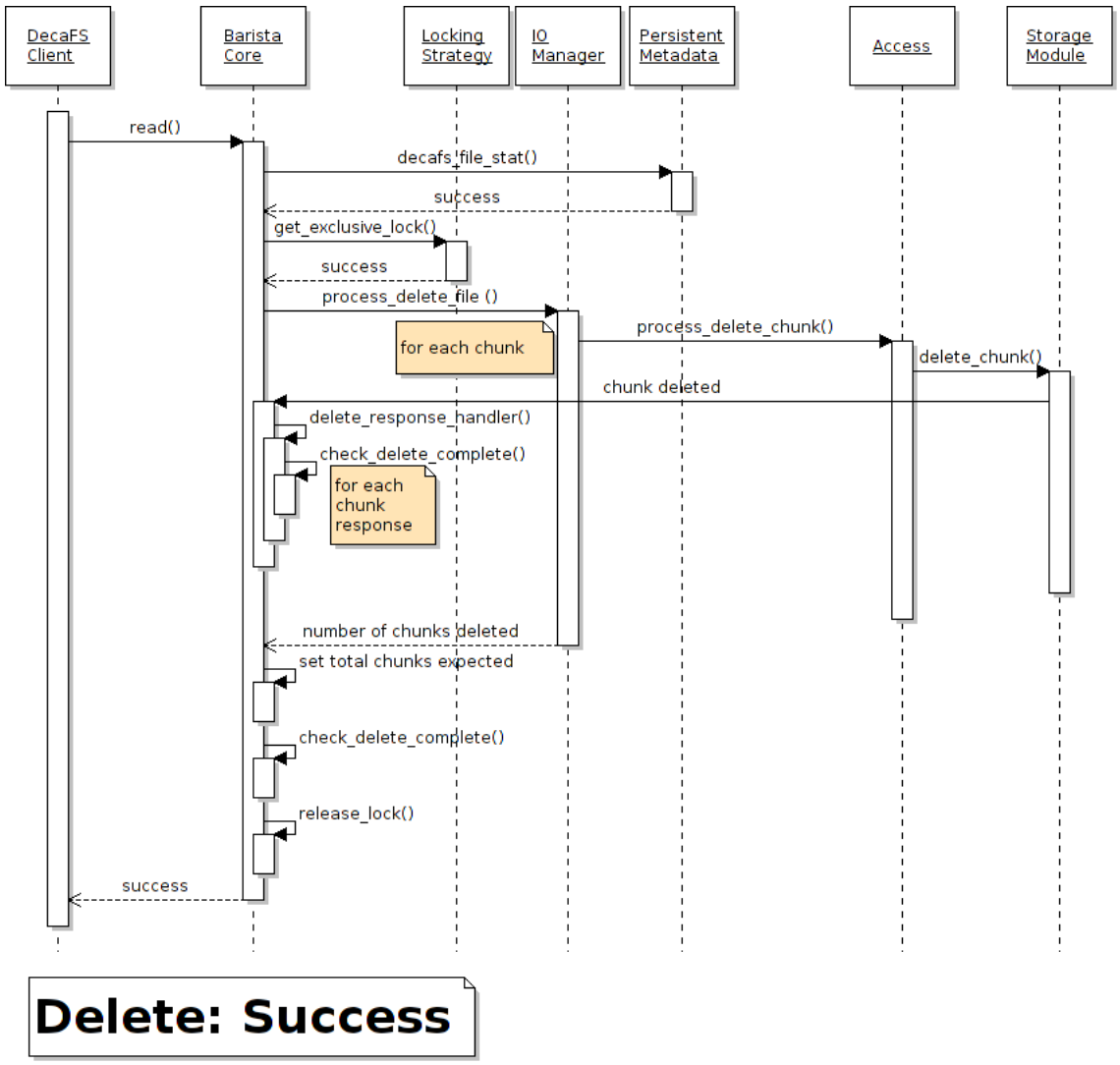


Figure 6.12: A successful call to delete.

## 6.7 Seek

Seek is another operation that occurs only on the Barista Layer. Components involved can be seen in Figure 6.13 and a successful execution is shown in Figure 6.14. A DecaFS Client's seek request arrives in Barista Core. Barista Core uses Volatile Metadata to ensure that the file cursor in question exists, and then sets the cursor to the new value under the calling DecaFS Client. Errors can occur if the file cursor does not exist, or the calling DecaFS Client is not the same as the client that opened the file. Error workflows can be seen in A.6.

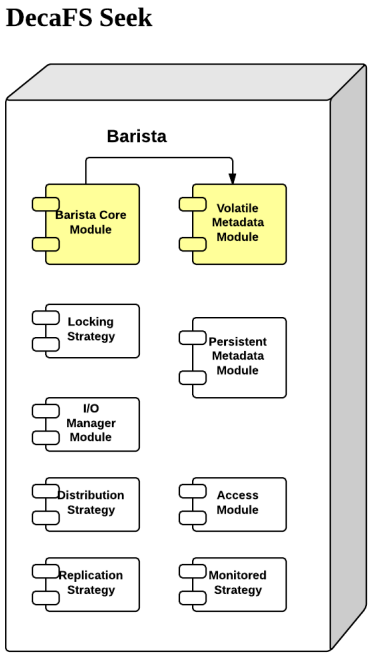


Figure 6.13: Components involved with a DecaFS seek call.

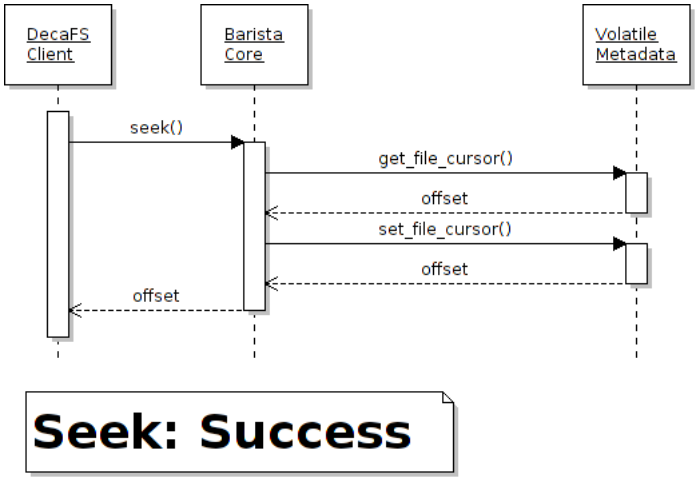


Figure 6.14: A successful call to seek.

## 6.8 Stat

Stat is an operation that occurs only on the Barista Layer. Components involved can be seen in Figure 6.15 and a successful call to stat is shown in Figure 6.16. All information about files stored in DecaFS is located in the Persistent Metadata Module. When a stat request arrives at Barista Core, information is queried from Persistent Metadata and returned to the DecaFS Client. An error may occur if the file in question does not exist, as seen in A.7.

### 6.8.1 Stat and Write Processing

Barista Core handles requests/responses one at a time. If a stat occurs after a write, the stat call will return information for the file **as if the write had already completed**, even though all chunks of the write request may not have been processed yet.

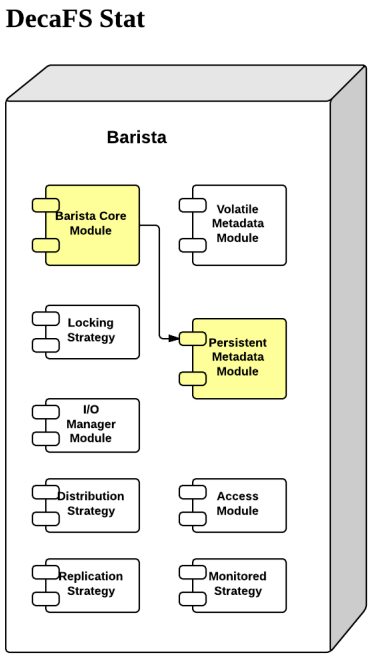


Figure 6.15: Components involved with a DecaFS stat call.

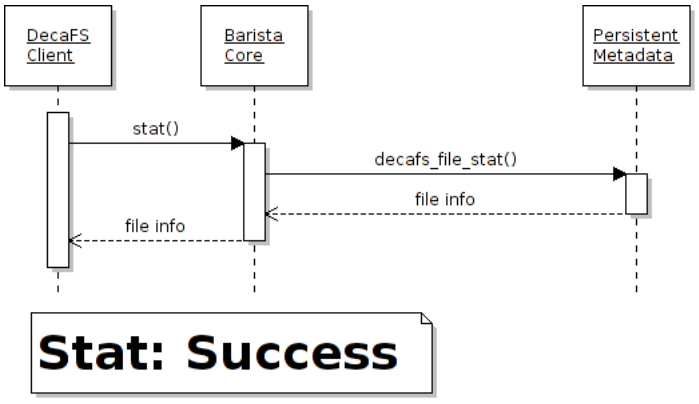


Figure 6.16: A successful call to stat.

# CHAPTER 7

# DecaFS Implementation

In this chapter we will discuss significant implementation details of the Barista Modules and Espresso Modules. Detailed information for the Network Layer can be found in Jeffrey Forrester's Master's Thesis [4]. My colleague's work describes the Network Layer in depth, as well as the system set-up for our use of DecaFS at Cal Poly.

The Barista Layer is implemented as a group of C++ classes and C-APIs that make up the nine Barista Modules. As discussed in Section 5.3, the main entry point for all calls to any module in the Barista Layer is Barista Core (Section 7.1). We discuss each module in detail, as well as the Espresso Storage Module in the following sections. Full function declarations and error codes are provided with the APIs for each module in Appendix B.

## 7.1 Barista Core

Barista Core handles requests from DecaFS Clients, and forwards information to other DecaFS Modules to fulfill them as discussed in CHAPTER 6. The full Barista Core API can be found in B.2 and an overview of the Client API from the Barista Layer can be seen in Table 7.1. In DecaFS, Barista Core Client Functions are called through the Network Layer from DecaFS Client Processes as discussed in my colleague's work [4]. The interaction between Barista Core and DecaFS Clients should not need to be altered by students. All DecaFS Client functions in Barista Core are void functions because client requests are asynchronous. Responses are processed in

separate functions, and results are sent over the Network back to DecaFS Clients once processing is complete.

| **Function** | **Description** |
|---|---|
| `void open_file (pathname, flags, DecaFS Client)` | Get a file descriptor for a DecaFS file **pathname**. A file is successfully opened for WRITE if no other process has the file open. A File is successfully opened for READ as long as other open requests for read come from the same DecaFS Client. |
| `void read_file (fd, count, DecaFS Client)` | Read **count** bytes from the open file **fd**. |
| `void write_file (fd, buf, count, DecaFS Client)` | Write **count** bytes from **buf** to the open file **fd**. |
| `void close_file (fd, DecaFS Client)` | Close an open file **fd**. |

| | |
|---|---|
| `void delete_file (pathname , DecaFS Client)` | Delete a file **pathname** as long as the file is not open by any DecaFS Client. |
| `void file_seek (fd, offset , whence, DecaFS Client)` | Move the file cursor, **fd** owned by **DecaFS Client** to the position specified by **offset** and **whence**. |
| `void file_stat (pathname, buf)` | Receive information from the file **pathname**. Information includes: file size, file id, stripe size, chunk size, and last access time. |
| `void open_directory ( pathname, DecaFS Client)` | Obtain a list of the files and directories that reside in **pathname** at the time of the call to open_directory (...). |

Table 7.1: Barista Core Client API

### 7.1.1 DecaFS Client Request State

In order to accommodate students that lack Network Experience, Barista Core maintains the state of each DecaFS Client Request. Barista Core also responds to DecaFS Clients when all chunked requests are fulfilled. This allows students to send requests to Espresso Nodes through their custom IO Manager Modules (5.3.5) without processing response packets sent back from Espresso Nodes to the Barista Node.

Barista Core monitors the state of each DecaFS Client request by maintaining maps for each request type that requires sub-processing on Espresso Nodes (read, write, and delete). For each request, Barista Core assigns a unique request id. Barista Core provides other Barista Modules with the request id for each striped request it sends to IO Manager as discussed in CHAPTER 6. IO Manager breaks down striped requests into chunked requests. Chunked requests must be sent through the Access Module (5.3.9) with this request id. The Network Layer stores the request id and ensures that Response Packets have the same id associated with them. Barista Core then uses the request id to organize Response Packets and determine when all responses to chunked requests have arrived. Once Barista Core has received all of the responses, the module sends a final response to the DecaFS Client.

#### 7.1.1.1 Request Information

All of Barista Core's state mappings require some common information for every request type. This information includes the following:

- **chunks_expected** number of chunked responses expected
- **file_id** the file id associated with the request
- **client** the DecaFS Client associated with the request

This general request information is managed through a struct called "request_info".

#### 7.1.1.2 Read

In addition to request_info, to manage state for a read request, Barista Core needs the following information:

- **fd** the open file descriptor for the read request
- **buf** the buffer where read results should be placed

As discussed briefly in Section 6.3, Barista Core requires that the IO Manager returns the number of chunks required for processing a striped request. Barista Core sums these chunk results from IO Manager, and sets **chunks_expected** to the sum once all stripes have been processed. After all stripes have been processed, Barista Core checks to see if all chunk responses have already arrived. If so, the final response packet is assembled. Otherwise, the module continues to process other requests.

When Barista Core receives read response packets, the packet is added to the read request state mapping. As Barista Core receives response packets, they are stored in the state mapping via a std::map, sorted by a chunk identifier made up of {file_id, stripe_id, chunk_id}. This chunk identifer key ensures that the responses are stored in logical order of file data. For every packet received, Barista Core checks to see if all chunk responses have arrived, if not, Barista Core continues to process other requests and responses. Once the final chunk response arrives, Barista Core assembles the final response packet.

This storage of response packets by chunk identifiers simplifies the process of assembling the final read buffer. The final packet assembly is a simple iteration through the map of packets. For each packet, chunk data is copied into the final read buffer, and the offset for the copy increases by the size of the chunk. After the map

is emptied, Barista Core sends the newly assembled read buffer and the number of bytes read to the DecaFS Client.

#### 7.1.1.3 Write

Write follows a similar process to read, but requires two separate request ids: one for primary write requests, and one for replica write requests. We require two separate ids since the DecaFS Client needs to be unaware of replication for transparency of the system. The separation of ids allows us to return the correct number of bytes written by a write request.

- **info** the request_info for primary chunked writes
- **replica_info** the request_info for replica writes
- **fd** the open file descriptor for the write request

Barista Core passes IO Manager both request ids with the calls to process each stripe for a write request. For each striped request, IO Manager must use return parameters to notify Barista Core of the number of primary chunk responses and replica responses created during the stripe's processing. Barista Core again, sums these response counts and stores the totals in the write request state mapping. Once all stripes have been processed, Barista Core checks to see if all primary chunk and replica responses have arrived. If so, Barista Core sends the final response packet to the DecaFS Client.

Barista Core does not save write response packets since write data does not need to be returned to the DecaFS Client. Instead, Barista Core sums the number of bytes written for each chunk and stores this count in the write state mapping for use when sending the final response. Both the primary write request id and the replica request id can be used to lookup state information in the write state mapping. As

write response packets arrive in Barista Core, response counts are incremented in the corresponding request (either **info** or **replica_info**). Once all responses to both the primary storage and the replica storage have been received, Barista Core sends the final response packet to the DecaFS Client.

#### 7.1.1.4 Delete

Delete requests do not need any additional info to maintain state, so the mapping uses the base information provided in request_info. Delete requests work in a similar manner to write, but are simplified since we do not need to differentiate between the deletion of primary storage and replica storage. Barista Core sends a request to IO Manager to delete all chunks associated with a specific file. IO Manager needs to return the number of chunked delete requests it sent out, this includes both primary and replica delete calls. Barista Core then waits for delete response packets to arrive, and, once all responses have been accounted for, sends the final response packet to the DecaFS Client.

### 7.1.2 Internal DecaFS Requests

In addition to handling DecaFS Client requests and responses, Barista Core facilitates the communication between DecaFS Modules. Modules within DecaFS may need to query one another to get information about the current state of the system. For example, Volatile Metadata contains information about which nodes are up at any given time, and an IO Manager may use this information to avoid sending requests to nodes that are down. Since many of our internals are implemented in C++, we use Barista Core to expose a C-API for student use of all Modules in the system. Barista Core manages instances of other modules, and calls through to these instances.

## 7.2 Persistent STL

DecaFS uses a custom library called "Persistent STL" to manage all metadata that needs to be written to disk. Persistent STL includes a PersistentMap and a PersistentSet. Both C++ classes are exposed for student use to manage additional metadata they may need in custom DecaFS Modules.

PersistentMap and PersistentSet adhere to the std::map and std::set interfaces respectively. In addition, open, flush, and close methods are exposed. PersistentMap and PersistentSet must be opened with a pathname that represents the file where the metadata should be stored. These classes may only be constructed with the default constructor, and a call to open must be made before use. If the PersistentMap and PersistentSet is not closed manually, the destructors will close and flush the file.

Both classes have similar implementations. They maintain an internal map <key, pointer to key/value>. Pointers in the internal map reference locations in the file that the PersistentMap/PersistentSet was opened with. These files are read directly into memory with the mmap system call [2], and the full data (key/value pairs) is stored in the file. PeristentMap/PersistentSet files work with static-sized keys and values.

Free space in the file is managed by a "free set" which maintains the freelist of pairs of offsets into the file. The free space is then the number of bytes between the first and second offset within the pair. An entry becomes deallocated when it is added to the free set, and the key is zeroed out in the file. Adjacent free spaces are combined to prevent fragmentation issues.

When a PersistentMap/PersistentSet is opened, if the pathname provided is a non-extant file, a new file is created and mmapped. If the file exists, it is opened and key/value data is read from disk. When a file is opened, all entries with zeroed out

keys are considered "free" and added to the free set in memory.

As mentioned previously, Persistent STL classes can be used by students who need a simple way to persist statically-sized collections of information. Within DecaFS we use Persistent STL in the Espresso Storage Module (Section 7.12), the Persistent Metadata Module (Section 7.3), and our implementation of the IO Manager Module (Section 7.6).

## 7.3 Persistent Metadata

The Persistent Metadata Module is responsible for storing information about which files are stored in DecaFS. Additionally, it maintains and assigns file ids within DecaFS. The Persistent Metadata uses PersistentMap (Section 7.2) to maintain a mapping of pathname to file id, and from file id to file metadata information. These PersistentMaps are recovered from disk when the Persistent Metadata Module is initialized on DecaFS start-up. Metadata information includes:

- **file_id** the id of the file
- **size** the size of the file
- **stripe_size** the stripe size the file was created with (immutable)
- **chunk_size** the chunk size the file was created with (immutable)
- **replica_size** the replica size the file was created with (immutable)
- **pathname** the name of the file
- **last_access_time** the timestamp for the last access to the file

Peristent Metadata exposes a C-API that allows for querying of metadata information and updating mutable fields, Table 7.2.

| Function | Description |
|---|---|
| `int get_num_files (DecaFS Client)` | Query for the number of files in DecaFS. |
| `int get_filenames (char * filenames_result, size, client)` | Query for the names of all files in DecaFS up to **size**. Result is stored in **filenames_result**. |
| `int decafs_file_stat ( file_id, buf, DecaFS Client)` | Get information about a file in DecaFS. Information includes: file size, file id, stripe size, chunk size, and last access time. |
| `int set_access_time ( file_instance, time, DecaFS Client)` | Set the access time for a file. |

| | |
|---|---|
| `int add_file (pathname, stripe_size, chunk_size, replica_size, time, DecaFS Client)` | Add metadata for a new file in DecaFS. |
| `int delete_file_contents ( file_id, DecaFS Client)` | Delete metadata for a file in DecaFS. |
| `int update_file_size ( file_id, size_delta, DecaFS Cilent)` | Alter the size of file **file_id** by **size_delta**. |

Table 7.2: Persistent Metadata External API

## 7.4 Volatile Metadata

The Volatile Metadata Module maintains metadata about the current state of the instance of DecaFS. This information includes the management of file cursors (the relation of client's processes to files) and system health (which nodes are up/down). These calls can be seen in Table 7.3 and Table 7.4.

| **Function** | **Description** |
|---|---|
| `uint32_t get_chunk_size()`<br><br>`void set_chunk_size (size)` | Get/Set the chunk size for the current instance of DecaFS (configurable in DecaFS start-up). |
| `uint32_t get_stripe_size()`<br><br>`void set_stripe_size (size )` | Get/set the stripe size for the current instance of DecaFS (configurable in DecaFS start-up). |
| `uint32_t get_num_espressos ()`<br><br>`int set_num_espressos (num )` | Get the number of espresso nodes expected to be connected in this instance of DecaFS (configurable in DecaFS start-up). |

| | |
|---|---|
| `uint32_t set_node_down( node)`<br><br>`uint32_t set_node_up (node )` | Changes a node's status to down/up in the current instance of DecaFS. |
| `bool is_node_up (node)` | Query for the status of a node (down/up). |
| `int get_active_node_count ()`<br><br>`active_nodes get_active_nodes ()` | Query for the number/node numbers of "active" (up) nodes in the current instance of DecaFS. |

Table 7.3: Volatile Metadata (System Health) External API

| Function | Description |
|---|---|
| `int new_file_cursor ( file_id, DecaFS Client)` | Get a new file cursor for the file specified by **file_id** under a specific **DecaFS Client**. |
| `int close_file_cursor (fd, DecaFS Client)` | Close an open file cursor **fd** under a specific textbfDecaFS Client. This call fails if the **DecaFS Client** provided did not open the file **fd**. |
| `int get_file_cursor (fd)` | Get the current offset within the file specified by **fd**. |
| `int set_file_cursor (fd, offset, DecaFS Client)` | Set the file cursor **fd** to **offset** under a specific **DecaFS Client**. This call fails if the **DecaFS Client** provided did not open the file **fd**. |
| `file_instance get_file_info (fd)` | Get information about the current instance of a file **fd**. This information includes: the file_id, the DecaFS Client who opened the file, and the current position of the cursor. |

| | |
|---|---|
| `uint32_t get_new_request_id()` | Get a new request id, unique to this instance of DecaFS. |

Table 7.4: Volatile Metadata (File Cursor) External API

## 7.5 Locking Strategy

Barista Core (Section 7.1) is responsible for the enforcement of locks within DecaFS since this module is the entry-point for DecaFS Client Requests in the system. However, Barista Core uses the Locking Strategy Module to obtain and release locks. The API Barista Core uses is described in Table 7.5.

An owning entity for a lock is a DecaFS Client, a single user running on a specific node. Only one entity can own a lock at a time. Within the DecaFS Client, multiple processes can own a shared lock **or** a single process can own an exclusive lock. We do not support upgrading locks, and all locks on a file must be released to change owning entities.

| Function | Description |
|---|---|
| `int get_exclusive_lock ( DecaFS Client, file_id)` | Attempt to get an exclusive lock (read/write) on a file **file_id**. |
| `int get_shared_lock ( DecaFS Client, file_id)` | Attempt to get a shared lock (read only) on a file **file_id**. |
| `int release_lock (DecaFS Client, file_id)` | Release a lock on a specific file, **file_id** under a specific **DecaFS Client**. |
| `int has_exclusive_lock ( DecaFS Client, file_id)` | Query whether or not this **DecaFS Client** has an exclusive lock on file **file_id** |
| `int has_shared_lock ( DecaFS Client, file_id)` | Query whether or not this **DecaFS Client** has a shared lock on file **file_id** |

Table 7.5: Locking Strategy External API

## 7.6 IO Manager

As discussed in 5.3.5 and in CHAPTER 6, the IO Manager Module is responsible for the conversion between striped and chunk-level requests in DecaFS. IO Manager is one of the main modules that students will need to implement to change DecaFS behavior.

We have provided a basic version of IO Manager as a sample that implements mirrored replication, and supports one node failures. Our IO Manager implementation uses two PersistentMaps (Section 7.2) to keep mappings from file chunk to the node it is stored on, and the replica for each chunk to the node it is stored on.

We define a file chunk to have the following properties:

- **file_id** the id of the file the chunk belongs to
- **stripe_id** the stripe within the file that the chunk belongs to
- **chunk_num** the order of this chunk within the stripe

For each chunk processed by IO Manager, we add our metadata needed to track chunks to these maps. Our IO Manager uses the Distribution Strategy (Section 7.7) and Replication Strategy (Section 7.8) modules to determine where each chunk and replica chunk should be sent. This way, students are able to re-write Distribution and Replication Modules, while using our Mirrored IO Manager.

Our IO Manager Module and our Distribution/Replication Strategy Modules query Volatile Metadata (Section 7.4). IO Manager will only send read/write requests to nodes that are up, and will re-try assigning a node to a chunk if the requested node from Distribution/Replication Strategy is down.

| Function | Description |
|---|---|
| `uint32_t process_read_stripe ( request_id, file_id, pathname, stripe_id, stripe_size, chunk_size, buf, offset, count)` | Break down a striped read request into corresponding chunked read request(s) and return the number of chunks needed to properly read the stripe. |
| `uint32_t process_write_stripe ( request_id, replica_request_id, chunks_written, replica_chunks_written, file_id, pathname, stripe_id, stripe_size, chunk_size, buf, offset, count)` | Break down a striped write request into corresponding chunked write request(s). Return the number of primary storage chunk writes and replica writes in **chunks_written** and **replica_chunks_written**. |

| | |
|---|---|
| `uint32_t process_delete_file ( request_id, file_id)` | Delete all chunks that belong to file **file_id**. |
| `int set_node_id (file_id, stripe_id, chunk_num, node_id)`<br><br>`int set_replica_node_id ( file_id, stripe_id, chunk_num, node_id)` | Set the storage location (**node_id**) for the chunk/replica specified by {**file_id**, **stripe_id**, **chunk_num**}. |
| `int get_node_id (file_id, stripe_id, chunk_num)`<br><br>`int get_replica_node_id ( file_id, stripe_id, chunk_num)` | Get the storage location (node id) for the chunk/replica specified by {**file_id**, **stripe_id**, **chunk_num**}. |

Table 7.6: IO Manager External API

## 7.7 Distribution Strategy

We implemented a basic Distribution Strategy for our IO Manager to use. This strategy sends odd chunks to node one and even chunks to node two. Students can implement more complex strategies.

| **Function** | **Description** |
|---|---|
| `int put_chunk (file_id, pathname, stripe_id, chunk_num)` | Determine which node a specific chunk should be sent to. |

Table 7.7: Distribution Strategy API

## 7.8 Replication Strategy

We implemented a basic Replication Strategy for our IO Manager to use. This strategy sends odd chunks to node three and even chunks to node four. Students can implement more complex strategies.

| **Function** | **Description** |
|---|---|
| `int put_replica (file_id, pathname, stripe_id, chunk_num)` | Determine which node a specific replica should be sent to. |

Table 7.8: Replication Strategy API

## 7.9 Node Failures for our Distribution/Replication Strategy

Our Basic Distribution and Replication Strategies are set up to support one node failures. We treat each pair of nodes as a "node group", so the Distribution Strategy uses node group one (nodes one and two) and the Replication Strategy uses node group two (nodes three and four).

### 7.9.1 Read

All chunk reads attempt to receive chunk data from the primary node first (Distribution node). If the node where the primary chunk data was written is down during

the time of the read, the replica chunk data is accessed instead.

### 7.9.2 Write

If a node is down at the time of a write, all data within the node group goes to one node. For example, if node two is down and a write occurs, all chunk data for the write goes to node one.

## 7.10 Access Module

Similar to Distribution and Replication Strategy Modules (Section 7.7, Section 7.8), the Access Module is implemented in its most basic form and can be extended by students. We provide a base Access Module that is the Barista Layer's hook into the Network to communicate with the Espresso Nodes. Chunked requests for the Espresso Nodes need to be sent through the Access Module with the API defined in Table 7.9.

| **Function** | **Description** |
|---|---|
| `ssize_t process_read_chunk (request_id, fd, file_id, node_id, stripe_id, chunk_num, offset, buf, count)` | Send a request to read **count** bytes into **buf** at chunk offset **offset** to a chunk defined by {**file_id**, **stripe_id**, **chunk_num**} to espresso node **node_id**. |
| `ssize_t process_write_chunk (request_id, fd, file_id, node_id, stripe_id, chunk_num, offset, buf, count)` | Send a request to write **count** bytes from **buf** at chunk offset **offset** to a chunk defined by {**file_id**, **stripe_id**, **chunk_num**} to espresso node **node_id**. |

| | |
|---|---|
| `ssize_t process_delete_chunk ( request_id, file_id, node_id, stripe_id, chunk_num)` | Send a request to delete a chunk defined by {**file_id**, **stripe_id**, **chunk_num**} to espresso node **node_id**. |

Table 7.9: Access API

## 7.11 Monitored Strategy

The Monitored Strategy Module allows students to implement their own system monitoring, metadata, and failure handlers. We provide a method strategy_startup () described in Table 7.10 for students to register their custom functions (Table 7.11) with Barista Core.

In strategy_startup () students should use the registration functions to register all functions they have implemented. Strategy_startup () will be called as part of the DecaFS System startup process, and is guaranteed to execute before DecaFS receives requests from DecaFS Clients.

One example of a task that can be implemented in this module is the cleanup of resources after a failed node recovers. For our implementation of IO Manager, when delete_file occurs, we send delete chunk requests for all chunks and replicas that reside on nodes that are up. However, if a node is down, we do not later go back and delete chunks/replicas that are stored on that node. To free up this space, we could implement a node_up_handler () to cleanup chunks that should have been deleted.

| **Function** | **Description** |
|---|---|
| `void strategy_startup ()` | A function called on during system startup for registration of monitoring modules. |

Table 7.10: Monitored Strategy API

| **Function** | **Description** |
|---|---|

| | |
|---|---|
| `void register_monitor_module (void (*f), timeout)` | Register a function **f** to be called every **timeout**. |
| `void register_node_failure _handler (void (* failure_handler)(node number))` | Register a function to be called when a node goes down. |
| `void register_node_up_handler (void (*handler)(node number))` | Register a function to be called when a node goes up. |

Table 7.11: Custom Strategy Registration API

## 7.12 Espresso Storage

As discussed in 5.4.1, the Espresso Storage Module is responsible for storing file data on disk.

The module uses three files to store file data:

1. Raw Data File
2. Metadata File
3. Free Extent Set File

Both the Metadata File and the Free Extent file use the Persistent STL (Section 7.2). The data file, is a raw, unordered, packed set of file chunks. The Metadata File and the Free Extent Set File use the following information with the Persistent STL libraries:

- data_descriptor
  - **file_id** the id of the file the chunk belongs to
  - **stripe_id** the stripe within the file that the chunk belongs to
  - **chunk_num** the order of this chunk within the stripe
- data_address
  - **offset** the offset into a raw storage file
  - **size** the size of the data stored at *offset*

The Metadata File is a PersistentMap that maps chunks (data_descriptor) to their offsets within the raw data file, and chunk sizing information (data_address). The Free Extents File is a collection of start/end pairs that signify free space.

### 7.12.1 read_chunk()

When a read_chunk() request arrives at the Espresso Storage Module, the module looks up the data_address of the chunk (data_descriptor) in the metadata PersistentMap. The metadata contains the offset into the raw data file for the chunk data, so the Espresso Storage Module can seek to the offset that was looked up and read the data.

### 7.12.2 write_chunk()

When Espresso Storage receives a write_chunk() request, the module scans the free extent set for an extent with sufficient size for the write. This scan occurs if the chunk does not exist, or if the existing chunk is not large enough. When an extent is chosen to place the chunk, it is shrunk down by the size of the write, or removed from the Free Extent Set and the write is issued. If a write exceeds the size of the chunk in its current location in the raw data file, the chunk grows in-place if possible. If not, the chunk is moved to a location in the file that can hold the new size of the chunk.

### 7.12.3 delete_chunk()

When a delete_chunk() request arrives at the Espresso Storage Module, the module adds freed data to the Free Extent Set. Adjacent free extents are merged.

| Function | Description |
|---|---|
| `ssize_t read_chunk (fd, file_id, stripe_id, chunk_num, offset, buf, count)` | Read **count** bytes at **offset** from chunk defined by {**file_id**, **stripe_id**, **chunk_num**} into **buf**. |
| `ssize_t write_chunk (fd, file_id, stripe_id, chunk_num, offset, buf, count)` | Write **count** bytes at **offset** from **buf** into chunk defined by {**file_id**, **stripe_id**, **chunk_num**}. |
| `int delete_chunk (fd, file_id, stripe_id, chunk_num)` | Delete chunk defined by {**file_id**, **stripe_id**, **chunk_num**}. |

Table 7.12: Espresso Storage External API

## 7.13 FUSE

FUSE is a "simple library API" that allows developers to implement a "fully functional filesystem in a userspace program [21]." We provide an implementation of the FUSE interface that allows students to mount DecaFS. Our FUSE Implementation is a wrapper around our DecaFS Client [4] that supports the following POSIX operations: getattr, mkdir, unlink, rmdir, open, read, write, close, opendir, readdir, closedir, create.

# CHAPTER 8

# Testing and Validation

In order to test for expected behavior, we utilized Google Test [12] and Google Mock [11] to verify modules and sub-systems (layers). Our formal testing efforts focused on the verification of each module, or sub-system (Barista, Network, Espresso) when the inter-module dependencies were too high to test the module independently. This section will describe our testing of the Barista Layer and the Espresso Layer. More information about Network Layer testing can be found in my colleague's Master's Thesis [4].

## 8.1 Google Test and Google Mock

Google Test is a C++ testing framework based on the xUnit architecture [12]. It supports "automatic test discovery, a rich set of assertions, user-defined assertions, death tests, fatal and non-fatal failures, value- and type-parameterized tests, various options for running the tests, and XML test report generation [12]." Google Test provides some unique features that help with debugging memory issues that surface only some of the time. For example, you can provide a flag that causes a certain test to repeat x number of times. Additionally, you can cause a test (or test suite) to automatically break on failure and launch a debugger [19]. Test filtering is also provided to allow developers to run a subset of tests based on matching a search string with test names [19].

Google Mock is a C++ mocking framework, compatible with Google Test, for

writing C++ Mock Classes [11]. Google Mock allows developers to "create mock classes trivially using simple macros" and use a "rich set of matchers and actions" for various expectations [11].

## 8.2 Espresso

Testing for Espresso Storage (Section 7.12) focuses on ensuring that chunks of any size can be stored properly. These test are implemented using Google Test. Espresso Storage tests rely on a fixture [12] that creates a new chunk file and both metadata files on start-up and removes these files during tear-down. This fixture enables the Espresso Storage Module to be tested without initialization of an entire Espresso node.

After start-up and file creation with the test fixture, unit tests exercise data storage logic within the Espresso Storage Module.

These tests address issues such as:

- Data can be written
- Data can be read
- Storage is compact
- Chunks may be reallocated if they become too large for their storage location
- Adjacent free blocks are merged

## 8.3 Barista

Testing the Barista Layer was more complex than the Espresso Layer due to dependencies between modules.

### 8.3.1 Independent Modules

Some of the modules in the Barista Layer were simple to test since they did not depend on the use of other modules. These modules were tested in a similar fashion to Espresso Storage (Section 8.2). We used Google Test to verify expected behavior with simple assertions.

Independent Modules and some issues their tests address are:

- Persistent STL (Section 7.2)
  - Test an instance of persistent classes for basic behavior as defined by std::map and std::set
- Persistent Metadata Module (Section 7.3)
  - Ensure that files may be added (fail if file exists or name is invalid)
  - Ensure that the number of files corresponds to the number of file successfully added
  - Ensure that metadata about stored files is correct
  - Ensure file size and access time may be modified
  - Ensure that files may be deleted
- Locking Strategy Module (Section 7.5)
  - Ensure that locks can be acquired
  - Ensure that locks contain the proper data about the lock owner
  - Ensure that locks can not be upgraded or downgraded
  - Ensure that multiple clients cannot hold any lock on the same file
  - Ensure that processes from the same client can hold shared locks

We consider Persistent Metadata an "independent module" even though it utilizes PersistentMap, since it can be recompiled and tested with std::map if desired.

### 8.3.2 Dependent Modules

Dependent modules were primarily tested manually a the system level. In order to provide more automatic testing of dependent modules, we used Google Mock [11] to implement Mock classes that specify the behavior of the dependencies. These mocks allow us to test each module independently, since dependent behavior is specified when defining the mock. After mocks are implemented for dependencies, the original module can be unit-tested with Google Test.

#### 8.3.2.1 Volatile Metadata

The most simple mock example is to facilitate the testing of the Volatile Metadata module (Section 7.4). Volatile Metadata is primarily an independent module, but it depends on Persistent Metadata (Section 7.3) for file cursor positions. This dependency is because we do not allow the file cursor's position to move past the end of a file for simplicity. In order to test Volatile Metadata completely independently, we need a mechanism for the module to stat a file in order to check the size of the file before moving a file cursor. We implemented a Mock Persistent Metadata Module that gives pre-defined file stat information for the Volatile Metadata test to use. This mocked information does not affect the validity of our test since we are simply attempting to verify that the file cursor cannot be moved past the end of the file (past file size).

#### 8.3.2.2 Other Mocks

We needed the following mocks to test dependent modules:

- Volatile Metadata (Section 7.4)
  - Persistent Metadata (Section 7.3) needs to be mocked (as described above) so that Volatile Metadata may stat information about a file
- Distribution Strategy (Section 7.7)
  - Volatile Metadata (Section 7.4) needs to be mocked so that this strategy module can query for system health (node up/down)
  - In student implementations, they may require mocks of other classes for more complex distribution strategies, such as Persistent Metadata (Section 7.3) for strategies that depend on file size (therefore requiring access to the stat function)
- Replication Strategy (Section 7.8)
  - Volatile Metadata (Section 7.4) needs to be mocked so that this strategy module can query for system health (node up/down)
  - In student implementations, they may require mocks of other classes for more complex replication strategies, such as Persistent Metadata (Section 7.3) for strategies that depend on file size (therefore requiring access to the stat function)

In our current implementation, the Access Module (Section 7.10) and the Monitored Strategy Module (Section 7.11) are implemented only as call-throughs, and do not provide any additional behavior. We have manually verified that these call-throughs occur with logs.

## 8.4 Data Storage

In order to test highly dependent modules Barista Core (Section 7.1) and IO Manager (Section 7.6), we added a stat function in addition to examining system behavior through logs. The Client Function, file_storage_stat() is used to verify the storage location(s) for chunks of a given file. This function is also useful in manual verification of Distribution/Replication strategies (Section 7.7 and Section 7.8).

**Sample Execution:**

This sample execution was run on a DecaFS configuration with a Barista Node and four Espresso Nodes. More information about system configuration and startup can be found in my colleague's work [4]. This version of DecaFS uses a mirrored IO Manager (Section 7.6) and basic Distribution/Replication strategies (Section 7.7 and Section 7.8).

*File Data:*

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
   23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
   41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58
   59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76
   77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94
   95 96 97 98 99 100
```

Write information:

- **size:** 291 bytes
- **stripe size:** 256 bytes
- **chunk size:** 128 bytes

- (stripe 1, chunk 1) - 128 bytes

  Primary Data: Node 1, Replica Data: Node 3

  1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46

- (stripe 1, chunk 2) - 128 bytes

  Primary Data: Node 2, Replica Data: Node 4

  47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 8

- (stripe 2, chunk 1) - 35 bytes

  Primary Data: Node 1, Replica Data: Node 3

  9 90 91 92 93 94 95 96 97 98 99 100

After the write was complete and the file was closed, we used a test DecaFS Client to call file_storage_stat().

*Storage Stat Output:*


```
{
  "file_id": 1
  "stripe_size": 256
  "chunk_size": 128
  "stripes": [
    {
      "stripe_id": 1
      "chunks": [
        {
          "chunk_num": 1
          "node": 1
```

```
            "replica_node": 3
          }
          {
            "chunk_num": 2
            "node": 2
            "replica_node": 4
          }
        ]
      }
      {
        "stripe_id": 2
        "chunks": [
          {
            "chunk_num": 1
            "node": 1
            "replica_node": 3
          }
        ]
      }
    ]
}
```

# CHAPTER 9

# Conclusions

## 9.1 Requirements

Here we will address how DecaFS meets our requirements from CHAPTER 4.

- REQ-1: *Students shall be able to develop architectural components of a distributed system*

  We presented DecaFS as a modular DFS so that students can develop modules (architectural components) of the system.

- REQ-2: *Students shall be able to build applications for a distributed system*

  DecaFS is mountable via FUSE, so student applications can use DecaFS to store files.

- REQ-3: *Students shall be able to change which node(s) data is stored on/recovered from.*

  Students may write Distribution and Replication Strategy Modules (Section 7.7, Section 7.8) to determine which node(s) data is stored on, without modifying other modules.

- REQ-4: *Students shall be able to change the replication policies of the system. The system should support no replication, mirroring, and some RAID [3] implementations.*

Students can write an IO Manager Module (Section 7.6) and Distribution and Replication Strategy Modules (Section 7.7, Section 7.8) to determine what types of replication the system uses. By default, we provide an IO Manager that supports mirroring. However, since the IO Manager has full control over storage at a stripe-level, students may implement various replication policies, including RAID. More information about IO Manager Replication Labs is available in my colleague's work [4].

- REQ-5: *The DFS shall be mountable with FUSE [21].*

  We provide a FUSE Client for DecaFS.

- REQ-6: *The system shall be able to tolerate at least one worker-node failure.*

  With our default IO Manager, one node failures are supported, since we provide mirrored replication. Students may also achieve one node failure support with a RAID Replication IO Manager.

- REQ-7: *DecaFS System Administrators shall be able to set the maximum possible file size for the DFS.*

  File size maximums are set through a configuration file for DecaFS start-up. This is discussed in my colleague's work [4].

- REQ-8: *DecaFS System Administrators shall be able to set the size (in bytes) of the stripes for each file, where a stripe is the maximum number of bytes of file data that are broken up into pieces and distributed for storage.*

  Stripe size is set through a configuration file for DecaFS start-up. This is discussed in my colleague's work [4].

- REQ-9: *DecaFS System Administrators shall be able to set the size (in bytes) of the chunks for each file, where a chunk is the maximum number of bytes of a stripe of file data stored at a time by one write to one storage node.*

Chunk size is set through a configuration file for DecaFS start-up. This is discussed in my colleague's work [4].

## 9.2 Discussion

As seen in Section 9.1, DecaFS was designed and implemented to meet our requirements. These requirements were created to ensure that DecaFS would be usable in the educational context. Due to the educational goals of our work, some common requirements such as performance, were not considered. These requirements were not considered in order to keep our implementation as simple as possible. Simplicity is an important goal of the system to ensure that start-up cost of learning the system APIs is a low as possible.

Our work has shown the feasibility of developing a DFS in modules. We hope that future use of DecaFS in Cal Poly classrooms will help students learn about distributed systems. Overall, DecaFS is a response to Hoganson, Google, IBM, and others [6, 13], providing an example of a mechanism for bringing distributed systems into the educational space.

# CHAPTER 10

# Future Work

We will improve this project in the upcoming academic year (2014-2015) in various areas. We hope the project can continue to grow as it is used in Cal Poly courses. All of our areas of improvement described in this section were not addressed throughout our current work due to time constraints.

## 10.1 Classroom Use

DecaFS was designed to be used in an educational setting. We have implemented versions of the modules that students will be expected to implement and provided samples of the types of projects that may be assigned to students [4]. However, we have not yet tested the system in an actual classroom setting. We hope that throughout the system's use at Cal Poly it can continue to improved based on student feedback.

## 10.2 Testing

As discussed in CHAPTER 8, we have provided basic testing to verify that DecaFS works as a minimally functional DFS. However, we would like to increase our testing methods for the future to ensure that DecaFS is behaving as expected.

### 10.2.1 Automated System Tests

As discussed in CHAPTER 8, we focused our testing efforts on Module and Sub-System testing. However, much of our effort to test DecaFS as a whole was through manual testing. We hope that future work on this project can develop a system test framework that can help verify the system's behavior automatically.

Additionally, a system-wide test suite would be beneficial because it could allow for automated testing of student submissions if the suite was altered to test for the behavior required from various assignments.

### 10.2.2 API Usability

Our work exposes APIs for students to use in developing individual modules or layers. Research has been done in the area of API Usability testing [8, 20]. Performing a usability study on our APIs could help make our system more usable for student work.

# BIBLIOGRAPHY


[1] Kosmosfs. `https://code.google.com/p/kosmosfs/`.

[2] Mmap (2). `http://man7.org/linux/man-pages/man2/mmap.2.html`.

[3] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. Raid: High-performance, reliable secondary storage. *ACM Comput. Surv.*, 26(2):145–185, June 1994.

[4] J. Forrester. Master's thesis, California Polytechnic State University, San Luis Obispo, CA United States, 2014.

[5] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, Oct. 2003.

[6] K. Hoganson. Computer science curricula in a global competitive environment. *J. Comput. Sci. Coll.*, 20(1):168–177, Oct. 2004.

[7] E. Levy and A. Silberschatz. Distributed file systems: Concepts and examples. *ACM Comput. Surv.*, 22(4):321–374, Dec. 1990.

[8] C. A. F. Marco Piccioni and B. Meyer. An empirical study of api usability.

[9] M. Ovsiannikov, S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly. The quantcast file system. *Proc. VLDB Endow.*, 6(11):1092–1101, Aug. 2013.

[10] Gemisus. About moosefs. `http://www.moosefs.org/`.

[11] Google, Inc. Google mock. `https://code.google.com/p/googlemock/`.

[12] Google, Inc. Google test. https://code.google.com/p/googletest/.

[13] Google, Inc. Google and ibm announce university initiative to address internet-scale computing challenges. http://googlepress.blogspot.com/2007/10/google-and-ibm-announce-university_08.html, Oct. 2007.

[14] International Business Machines Corporation and others. Openafs. http://www.openafs.org/.

[15] The Apache Software Foundation. Hdfs architecture guide. http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.

[16] University of Pittsburgh. Andrew file system (afs). http://technology.pitt.edu/network-web/hosting-timesharing/afs.html.

[17] Zuse Institute Berlin. Xtreemfs. http://www.xtreemfs.org/.

[18] Z. Ruan and W. F. Tichy. Performance analysis of file replication schemes in distributed systems. In *Proceedings of the 1987 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '87, pages 205–215, New York, NY, USA, 1987. ACM.

[19] A. Sen. A quick introduction to the google c++ testing framework. http://www.ibm.com/developerworks/aix/library/au-googletestingframework.html.

[20] J. Stylos and B. A. Myers. The implications of method placement on api learnability. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 105–112, New York, NY, USA, 2008. ACM.

[21] M. Szeredi. Filesystem in userspace. http://fuse.sourceforge.net/.

[22] A. Wachsmann. Afs—a secure distributed filesystem, part iii. *Linux J.*, 2005(132):8–, Apr. 2005.

[23] J. Wein, K. Kourtchikov, Y. Cheng, R. Gutierez, R. Khmelichek, M. Topol, and C. Sherman. Virtualized games for teaching about distributed systems. *SIGCSE Bull.*, 41(1):246–250, Mar. 2009.

# APPENDIX A

## Workflows for Failures

### A.1 Open Failures

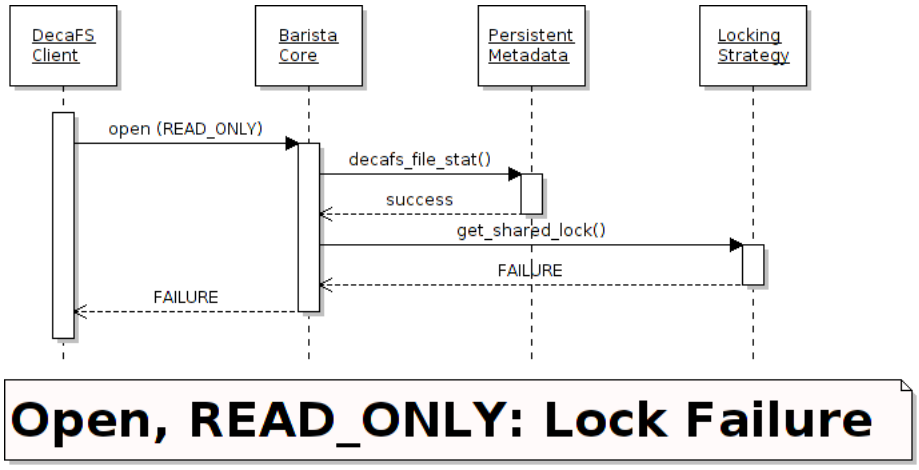


Figure A.1: A failed call to open due to the DecaFS Client being unable to obtain a shared lock on the file.

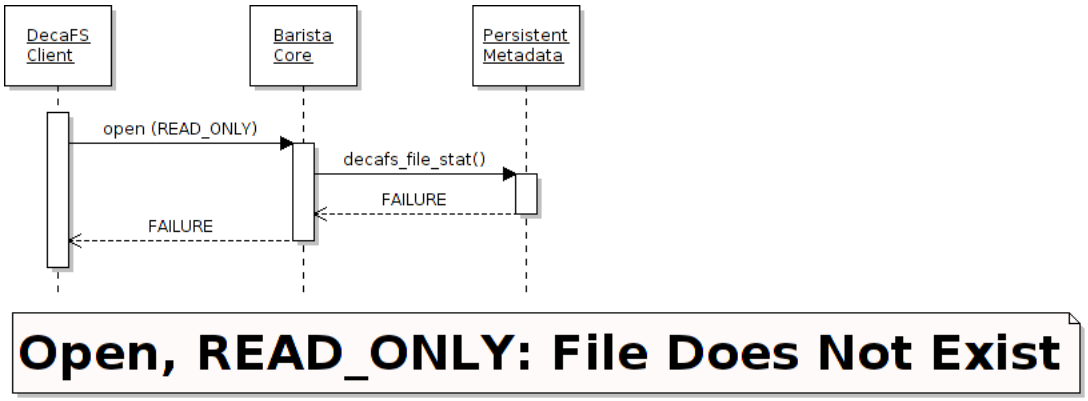


Figure A.2: A failed call to open due to the file not being found.

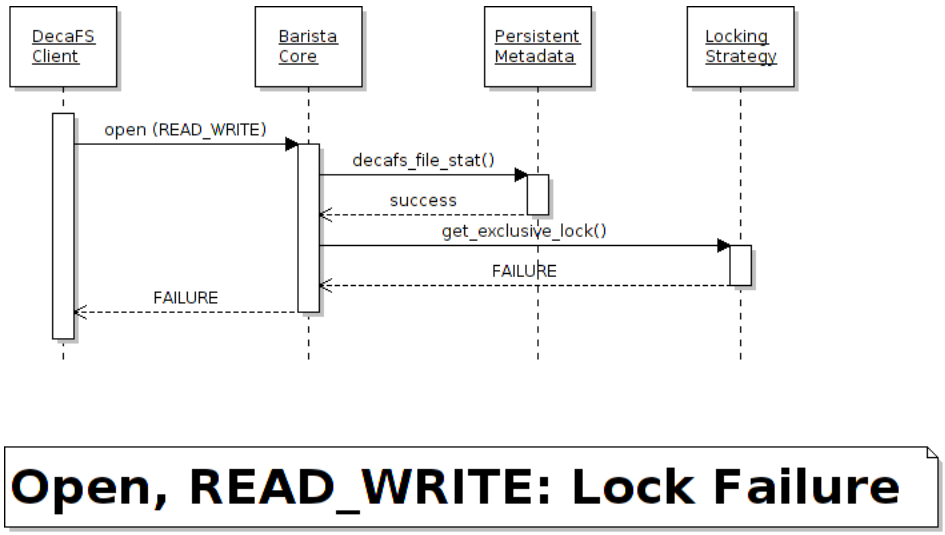


Figure A.3: A failed call to open due to the DecaFS Client being unable to obtain an exclusive lock on the file.

## A.2 Read Failures

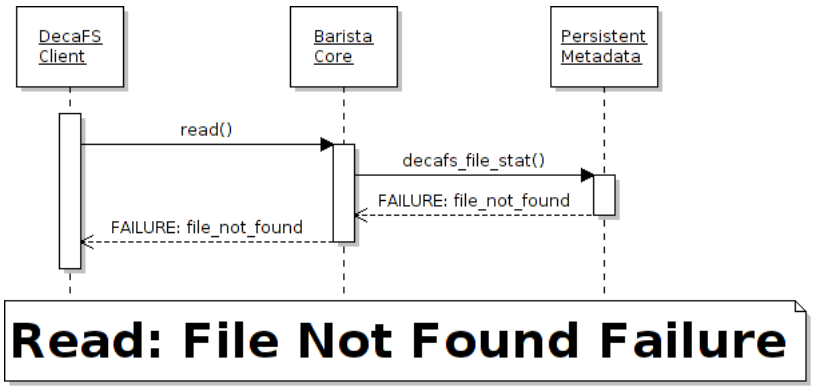


Figure A.4: A failed call to read due to the file being non-existent.

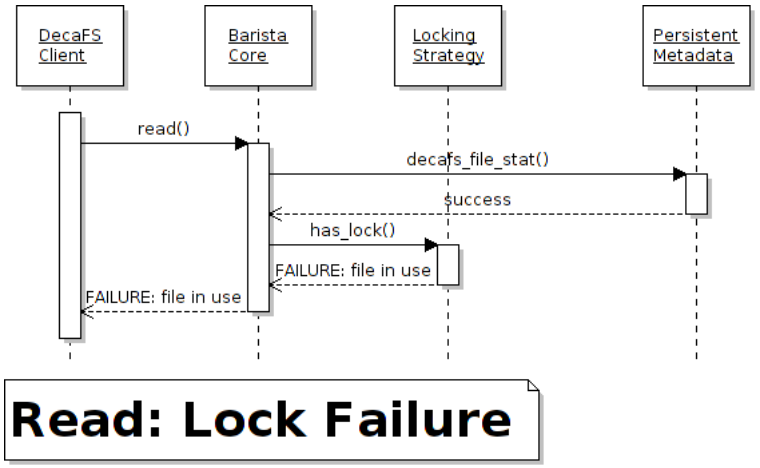


Figure A.5: A failed call to read because the DecaFS Client does not have a suitable lock on the file.

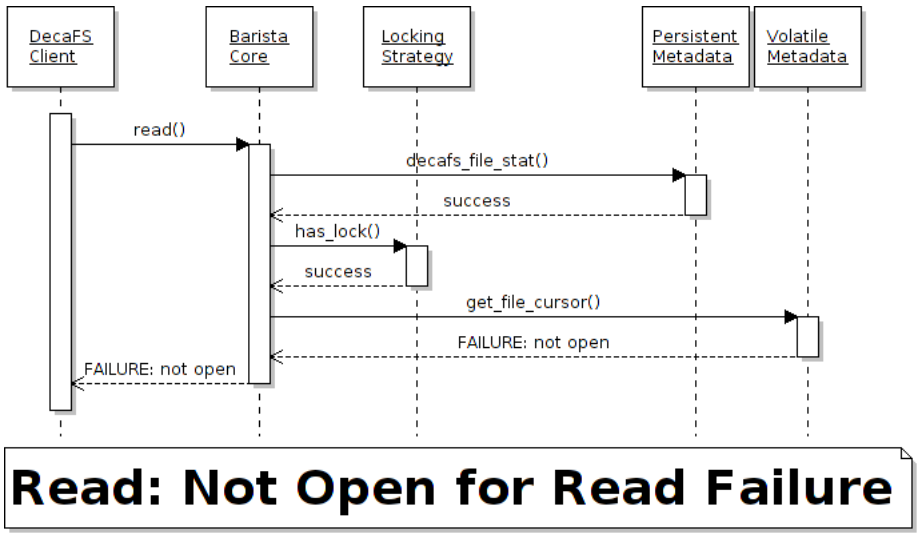


Figure A.6: A failed call to read due to the file cursor not existing.

## A.3 Write Failures

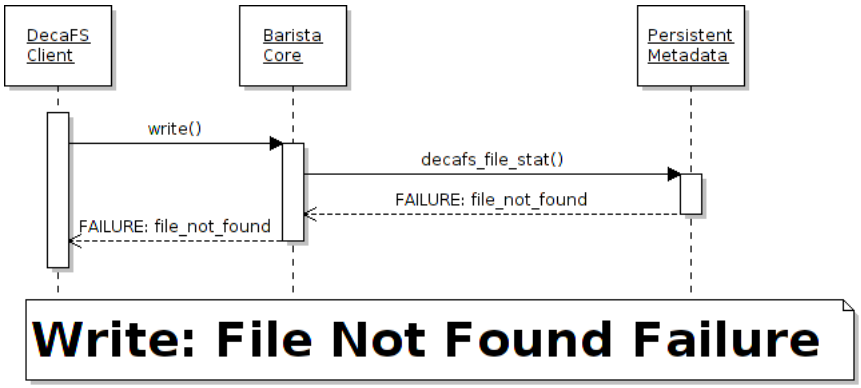


Figure A.7: A failed call to read due to the file being non-existent.

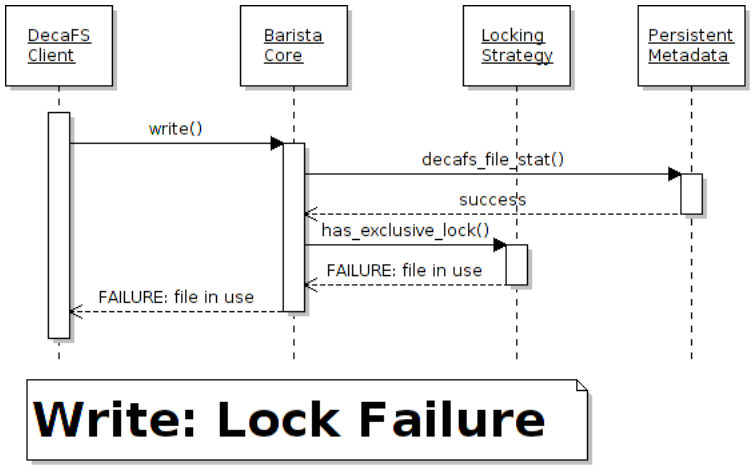


Figure A.8: A failed call to write because the DecaFS Client does not have an exclusive lock on the file.

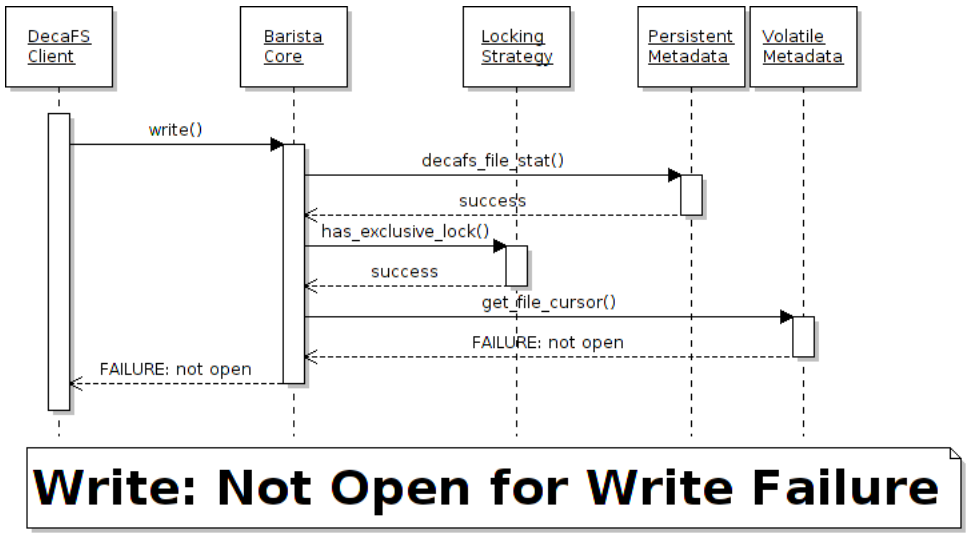


Figure A.9: A failed call to write due to the file cursor not existing.

## A.4 Close Failures

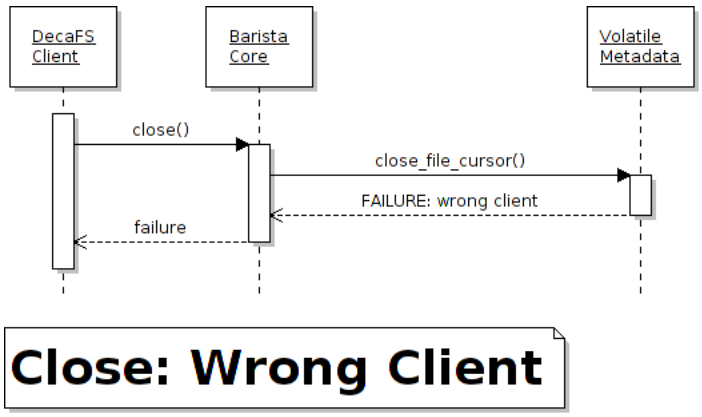


Figure A.10: A failed call to close due to the calling DecaFS Client being different than the DecaFS Client that opened the file.

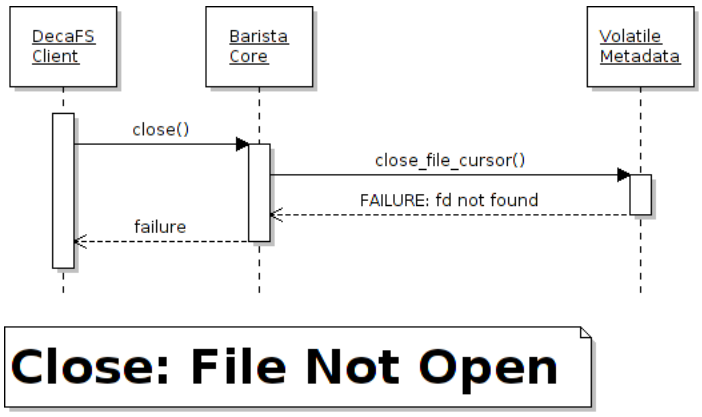


Figure A.11: A failed call to close due to the file not being open in the first place.

## A.5 Delete Failures

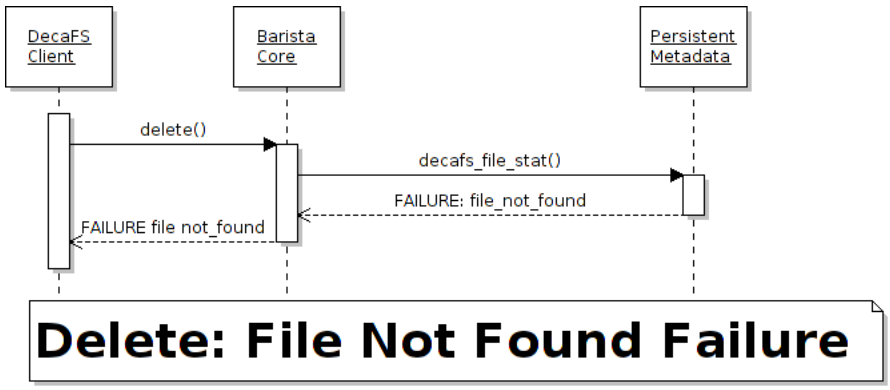


Figure A.12: A failed call to delete due to the file's non-existence.

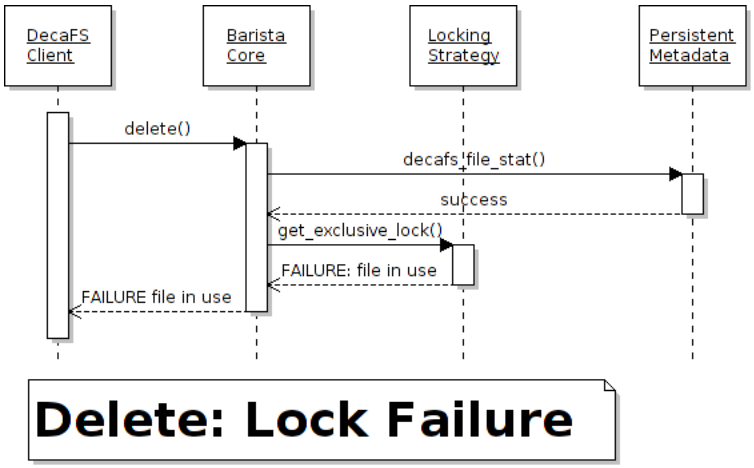


Figure A.13: A failed call to delete due to the file being in use.

## A.6 Seek Failures

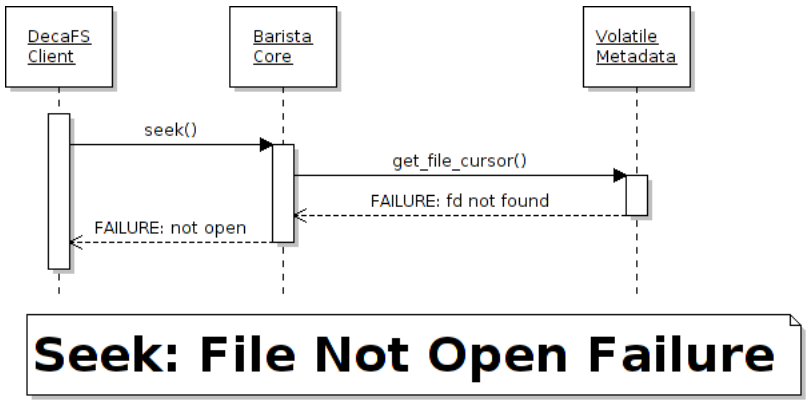


Figure A.14: A failed call to seek due to the file descriptor being invalid.

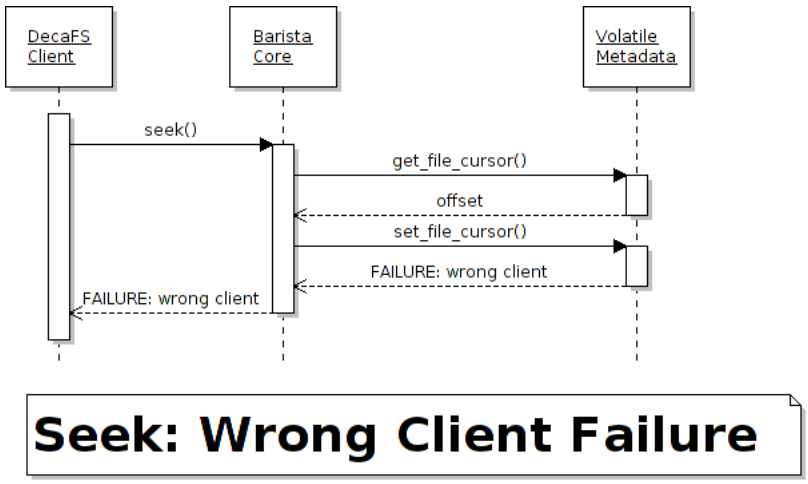


Figure A.15: A failed call to seek due to the calling DecaFS Client differing from the DecaFS Client associated with the file descriptor in question.

## A.7 Stat Failures

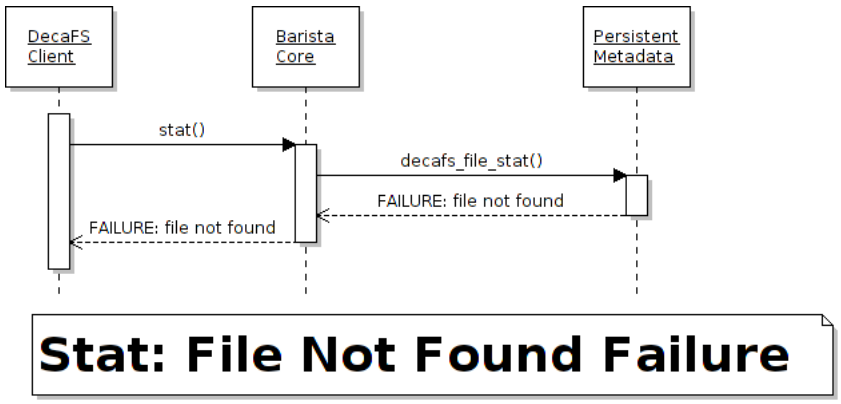


Figure A.16: A failed call to stat due to the file not existing.

# APPENDIX B

# APIs

## B.1 DecaFS Types

```
1  /*
2   * DIR equivalent for DecaFS
3   */
4  struct decafs_dir {
5    int current;
6    int total;
7    struct decafs_dirent *entries;
8
9    decafs_dir(int current, int total, decafs_dirent*
       entries) :
10    current(current), total(total), entries(entries) {}
11   decafs_dir(int total, decafs_dirent* entries) :
       decafs_dir(0, total, entries) {}
12
13 };
14
15 struct decafs_dirent {
16   uint32_t file_id; // id unknown for directories
17   unsigned char d_type; // 'f' file, 'd' directory
```

```
18   char d_name[256]; // entry name
19
20   decafs_dirent(uint32_t file_id, unsigned char d_type,
        char* name) :
21    file_id(file_id), d_type(d_type) {
22
23     memcpy(d_name, name, strlen(name) + 1);
24   }
25 };
26
27 /*
28  * Stores information about a specific instance of an open
        file in
29  * DecaFS.
30  */
31 struct file_instance {
32   struct client client_id;
33   uint32_t file_id;
34   uint32_t offset;
35   file_instance(): client_id (client()), file_id (0),
        offset (0) {}
36   file_instance (struct client client, uint32_t file_id,
        uint32_t offset) {
37     this->client_id = client;
38     this->file_id = file_id;
39     this->offset = offset;
40   }
```

```
41
42   bool operator ==(const file_instance & other) const {
43     return (this->client_id == other.client_id &&
44             this->file_id == other.file_id);
45   }
46
47   bool operator <(const file_instance &other) const {
48     if (this->client_id != other.client_id) {
49       return this->client_id < other.client_id;
50     }
51     return (this->file_id < other.file_id);
52   }
53 };
54
55 /*
56  * Distinctly idenfies a chunk of a file.
57  */
58 struct file_chunk {
59   uint32_t file_id;
60   uint32_t stripe_id;
61   uint32_t chunk_num;
62
63   bool operator ==(const file_chunk & other) const {
64     return (this->file_id == other.file_id &&
65             this->stripe_id == other.stripe_id &&
66             this->chunk_num == other.chunk_num);
67   }
```

```
68
69   friend bool operator <(const file_chunk &left, const
        file_chunk &right) {
70     if (left.file_id != right.file_id) {
71       return left.file_id < right.file_id;
72     }
73     if (left.stripe_id != right.stripe_id) {
74       return left.file_id < right.stripe_id;
75     }
76     return left.chunk_num < right.chunk_num;
77   }
78 };
79
80 /*
81  *  Storage Information about one file in DecaFS.
82  */
83 struct decafs_file_stat {
84   uint32_t file_id; /* DecaFS file id for the file. */
85   uint32_t size; /* Size of the file in bytes */
86   uint32_t stripe_size;
87   uint32_t chunk_size;
88   uint32_t replica_size;
89   struct timeval last_access_time;
90 };
91
92 struct ip_address {
93   char addr[IP_LENGTH];
```

```
94    ip_address() : addr {'\0'} {};
95    ip_address(char *addr) {
96      strcpy (this->addr, addr);
97    }
98
99    bool operator ==(const ip_address & other) const {
100     return (strcmp (this->addr, other.addr) == 0);
101   }
102
103   bool operator !=(const ip_address & other) const {
104     return !operator==(other);
105   }
106
107   bool operator <(const ip_address & other) const {
108     return (strcmp (this->addr, other.addr) <= 0);
109   }
110 };
111
112 struct client {
113   struct ip_address ip;
114   uint32_t user_id;
115   ConnectionToClient *ctc;
116
117   client() : ip (ip_address()), user_id (0), ctc (NULL)
        {};
118   client(struct ip_address ip, uint32_t user_id,
        ConnectionToClient *ctc) :
```

```
119     ip(ip), user_id (user_id), ctc (ctc) {}
120
121   bool operator ==(const client & other) const {
122     return (this->ip == other.ip &&
123             this->user_id == other.user_id &&
124             this->ctc == other.ctc);
125   }
126
127   bool operator !=(const client & other) const {
128     return !operator==(other);
129   }
130
131   bool operator <(const client & other) const {
132     return ((this->ip < other.ip) ? true :
133               (this->user_id < other.user_id) ? true :
134                  (this->ctc < other.ctc) ? true : false);
135   }
136 };
137
138 struct active_nodes {
139   uint32_t node_numbers[NUM_ESPRESSO];
140   uint32_t active_node_count;
141 };
```

## B.2 Barista Core API

```
1 struct request_info {
```

```
2   uint32_t chunks_expected;
3   uint32_t chunks_received;
4   uint32_t file_id;
5   struct client client;
6
7   request_info() : chunks_expected (0), chunks_received
       (0), file_id (0) {}
8   request_info (struct client client, uint32_t file_id) {
9       this->chunks_expected = 0;
10      this->chunks_received = 0;
11      this->file_id = file_id;
12      this->client = client;
13  }
14};
15
16struct read_buffer {
17  int size;
18  uint8_t *buf;
19
20  read_buffer() : size (0), buf (NULL) {}
21  read_buffer (int size, uint8_t *buf) {
22    if (size > 0) {
23      this->buf = (uint8_t *)malloc(size);
24      memcpy (this->buf, buf, size);
25      this->size = size;
26    }
27    else {
```

```
28        this->size = 0;
29        this->buf = NULL;
30      }
31    }
32    ~read_buffer () {
33      if (size > 0) {
34        free(this->buf);
35      }
36    }
37  };
38
39  struct read_request_info {
40    struct request_info info;
41    int fd;
42    uint8_t *buf;
43    std::map<struct file_chunk, struct read_buffer*>
        response_packets;
44
45    read_request_info() : info (request_info()), fd (0) {}
46    read_request_info (struct client client, uint32_t
        file_id, int fd,uint8_t *buf) {
47      this->info = request_info (client, file_id);
48      this->fd = fd;
49      this->buf = buf;
50    }
51  };
52
```

```
53 struct write_request {
54   uint32_t request_id;
55   uint32_t replica_request_id;
56
57   bool operator <(const write_request &other) const {
58     if (this->request_id != other.request_id) {
59       return this->request_id < other.request_id;
60     }
61     return (this->replica_request_id < other.
          replica_request_id);
62   }
63 };
64
65 struct write_request_info {
66   struct request_info info;
67   struct request_info replica_info;
68   int fd;
69   int count;
70
71   write_request_info() : info (request_info()),
        replica_info (request_info()), fd (0), count (0) {}
72   write_request_info (struct client client, uint32_t
        file_id, int fd) {
73     this->info = request_info (client, file_id);
74     this->replica_info = request_info (client, file_id);
75     this->fd = fd;
76     this->count = 0;
```

```
77   }
78 };
79
80 extern "C" const char *get_size_error_message (const char
     *type, const char *value);
81
82 extern "C" void exit_failure (const char *message);
83
84 /*
85  * Initialize barista core
86  */
87 extern "C" void barista_core_init (int argc, char *argv[])
     ;
88
89 /*
90  *  Open a file for read or write access.
91  *
92  * Flags:
93  *   O_RDONLY open a file for reading
94  *   O_RDWR open a file for both reading and writing
95  *   O_APPEND start the file cursor at the end of the file
96  *
97  *  @post
98  *     open_file sends the file id for the newly opened
     file (non-zero)
99  *     to the client or FILE_IN_USE if the proper lock
     cannot be obtained
```

```
100  */
101 extern "C" void open_file (const char *pathname, int flags
      , struct client client);
102
103 /*
104  *  opens a directory stream corresponding to the
       directory name.
105  */
106 extern "C" void open_dir (const char* name, struct client
      client);
107
108 /*
109  *  If the process has a lock on the file, complete the
       read.
110  *  Translates read request into chunks of requests to
       Espresso
111  *  nodes.
112  */
113 extern "C" void read_file (int fd, size_t count, struct
      client client);
114
115 /*
116  * Aggregates the read_file futures and determines when
       the read is complete.
117  * Upon completion of a read, this function returns read
       information to the
118  * Network Layer.
```

```
119  */
120 extern "C" void read_response_handler (ReadChunkResponse *
      read_response);
121
122 /*
123  *  If the process has an exclusive lock on the file,
       complete the
124  *  write.
125  *  Translate write requests into chunks of requests to
       Espresso
126  *  nodes.
127  */
128 extern "C" void write_file (int fd, const void *buf,
      size_t count, struct client client);
129
130 /*
131  * Aggregates the write_file futures and determines when
       the write is complete.
132  * Upon completion of a write, this function returns write
        information to the
133  * Network Layer.
134  */
135 extern "C" void write_response_handler (WriteChunkResponse
       *write_response);
136
137 /*
138  *  Release locks associate with a fd.
```

```
139  */
140 extern "C" void close_file (int fd, struct client client);
141
142 /*
143  *  Removes a file from DecaFS.
144  *  @ return >= 0 success, < 0 failure
145  */
146 extern "C" void delete_file (char *pathname, struct client
        client);
147
148 /*
149  * Aggregates the delete_file futures and determines when
        the delete is complete.
150  * Upon completion of a delete, this function returns
        delete information to the
151  * Network Layer.
152  */
153 extern "C" void delete_response_handler (
      DeleteChunkResponse *delete_response);
154
155 /*
156  * Moves the file cursor to the location specificed by
        whence, plus offset
157  * bytes.
158  *
159  * If the whence and offset cause the cursor to be set
        past the end of the file
```

```
160  * it will be set to the end of the file.
161  *
162  * whence:
163  *   SEEK_SET move to offset from the beginning of the
       file
164  *   SEEK_CUR move to offset from the current location of
       the fd
165  *   SEEK_END move to end of file
166  *
167  * client will receive the cursor's new location on
       success and < 0 on failure
168  *
169  */
170 extern "C" void file_seek (int fd, uint32_t offset, int
      whence, struct client client);
171
172 /*
173  *  Fills struct stat with file info.
174  */
175 extern "C" void file_stat (const char *path, struct stat *
      buf);
176 extern "C" void file_fstat (int fd, struct stat *buf);
177
178 /*
179  * Get the storage and replica storage information for a
       file.
180  */
```

```
181 extern "C" void file_storage_stat (const char *path,
      struct client client);
182
183 /*
184  *  Collects information about a mounted filesystem.
185  *  path is the pathname of any file within the mounted
186  *  filesystem.
187  */
188 extern "C" void statfs (char *pathname, struct statvfs *
      stat);
189
190 /*
191  *  Move an existing chunk to a different Espresso node in
        the system.
192  */
193 extern "C" void move_chunk (const char* pathname, uint32_t
       stripe_id, uint32_t chunk_num, uint32_t dest_node,
      struct client client);
194 extern "C" void fmove_chunk (uint32_t file_id, uint32_t
      stripe_id, uint32_t chunk_num, uint32_t dest_node,
      struct client client);
195
196 /*
197  *  Move a chunk s replica to a different Espresso node
      in the system.
198  */
```

```
199 extern "C" void move_chunk_replica (const char* pathname,
      uint32_t stripe_id, uint32_t chunk_num, uint32_t
      dest_node, struct client client);
200 extern "C" void fmove_chunk_replica (uint32_t file_id,
      uint32_t stripe_id, uint32_t chunk_num, uint32_t
      dest_node, struct client client);
```

### B.3 Persistent Metadata API

```
1  /*
2   *  Return the number of files that exist in DecaFS.
3   */
4  extern "C" int get_num_files (struct client client);
5
6  /*
7   *  Provide a list of filenames that exist in DecaFS.
8   *  filenames must have space to hold the number of
       filenames
9   *  returned by get_num_files().
10  *  @param size number of file names of length
       MAX_FILENAME_LENGTH
11  *       that fit in filenames.
12  *  @return the number of files stored in filenames array
       in
13  *       alphabetical order
14  */
```

```
15 extern "C" int get_filenames (char *filenames[
     MAX_FILENAME_LENGTH], int size, struct client client);
16
17 /*
18   *  Fill in system stat structure with information
19  *  about one file.
20  *  @ return 0 on success
21  *           FILE_NOT_FOUND on failure
22  */
23
24 extern "C" int decafs_file_sstat (char *pathname, struct
     decafs_file_stat *buf, struct client client);
25 extern "C" int decafs_file_stat (uint32_t file_id, struct
     decafs_file_stat *buf, struct client client);
26
27 /*
28  *  Fill in system stat structure with information
29  *  about entire mounted DecaFS.
30  */
31 extern "C" int decafs_stat (char *pathname, struct statvfs
      *buf, struct client client);
32
33 /*
34  *  Updates the access time of the file.
35  *  @ return 0 on success
36              FILE_NOT_FOUND on error
37  */
```

```
38 extern "C" int set_access_time (file_instance inst, struct
       timeval time, struct client client);
39
40 /*
41  *  Add a file to the DecaFS metadata.
42  *  @return file_id on success
43  *          FILE_EXISTS if filename already exists in
       DecaFS
44  *          FILENAME_INVALID if filename is too long
45  */
46 extern "C" int add_file (char *pathname, uint32_t
      stripe_size, uint32_t chunk_size, uint32_t replica_size
      , struct timeval time, struct client client);
47
48 /*
49  * Removes a file from DecaFS metadata.
50  * @ return 0 on success
51  *          FILE_NOT_FOUND on error
52  */
53 extern "C" int delete_file_contents (uint32_t file_id,
      struct client client);
54
55 /*
56  *  Update the size (add or remove bytes to a file) of an
       existing file.
57  *  @return the size of the new file on success
58  *          FILE_NOT_FOUND on failure
```

```
59  */
60 extern "C" int update_file_size (uint32_t file_id, int
     size_delta, struct client client);
```

## B.4 Volatile Metadata API

```
1  /*
2   * Returns the chunk size that is set for this instance of
       DecaFS.
3   * If chunk size has not been set yet, this function
      returns 0.
4   */
5  extern "C" uint32_t get_chunk_size ();
6
7  /*
8   * Sets the chunk size for this instance of DecaFS.
9   * If chunk size has already been set, SIZE_ALREADY_SET is
       returned.
10  * If the chunk size provided is an invalid size
      SIZE_INVALID is returned.
11  */
12 extern "C" int set_chunk_size (uint32_t size);
13
14 /*
15  * Returns the stripe size that is set for this instance
      of DecaFS.
```

```
16  * If stripe size has not been set yet, this function
       returns 0.
17  */
18 extern "C" uint32_t get_stripe_size ();
19
20 /*
21  * Sets the stripe size for this instance of DecaFS.
22  * If stripe size has already been set, SIZE_ALREADY_SET
       is returned.
23  * If the stripe size provided is an invalid size
       SIZE_INVALID is returned.
24  */
25 extern "C" int set_stripe_size (uint32_t size);
26
27 /*
28  * Returns the number of espresso nodes that should be
       connected for this
29  * instance of DecaFS.
30  */
31 extern "C" uint32_t get_num_espressos ();
32
33 /*
34  * Sets the number of espresso nodes to expect for this
       instance of DecaFS.
35  * If the number of espressos is already set,
       SIZE_ALREADY_SET is returned.
36  */
```

```
37 extern "C" int set_num_espressos (uint32_t num_espressos);
38
39 /*
40  * Set the node with the unique node_number to be "down"
       in the instance
41  *   of DecaFS.
42  * @return V_META_SUCCESS on success
43  *         NODE_NUMBER_NOT_FOUND on failure
44  */
45 extern "C" uint32_t set_node_down (uint32_t node_number);
46
47 /*
48  * Set the node with the unique node_number to be "down"
       in the instance
49  *   of DecaFS.
50  * @return V_META_SUCCESS on success
51  *         NODE_NOT_FOUND on failure
52  */
53 extern "C" uint32_t set_node_up (uint32_t node_number);
54
55 /*
56  * Determines whether or not a specific node is "up"
57  */
58 extern "C" bool is_node_up (uint32_t node_number);
59
60 /*
61  * Returns the number of active nodes.
```

```
62  */
63 extern "C" int get_active_node_count();
64
65 /*
66  * Gives the "state" of the system.
67  * Returns an active_nodes struct that represents the node
        numbers active
68  *  in the current instance of DecaFS.
69  */
70 extern "C" struct active_nodes get_active_nodes ();
71
72 /*
73 *   Start a new file cursor if one doesn t exist already
       .
74 *  @return the fd
75 */
76 extern "C" int new_file_cursor (uint32_t file_id, struct
       client client);
77
78 /*
79 *   Remove a file cursor for an open instance of a file.
80 *   @return id of the file closed on success
81 *   @return INSTANCE_NOT_FOUND if fd does not exist
82 *   @return WRONG_CLIENT if the client doesn't match the
       client who
83 *           opened the file
84 */
```

```
85 extern "C" int close_file_cursor (uint32_t fd, struct
     client client);
86
87 /*
88  * Provides information about the cursor for an instance
      of an open
89  *   file.
90  *  @return the current byte offset for a given fd
91  *          if the fd does not exist, INSTANCE_NOT_FOUND
      is returned.
92  */
93 extern "C" int get_file_cursor (uint32_t fd);
94
95 /*
96  *   Set the cursor for an instance of an open file.
97  *  @return the current byte offset for a given fd
98  *          if the fd does not exist, INSTANCE_NOT_FOUND
      is returned.
99  */
100 extern "C" int set_file_cursor (uint32_t fd, uint32_t
     offset, struct client client);
101
102 /*
103  * Find the file_instance associated with a given fd.
104  */
105 extern "C" struct file_instance get_file_info (uint32_t fd
     );
```

```
106
107 /*
108  * Get a new request id for a client request.
109  */
110 extern "C" uint32_t get_new_request_id();
```

## B.5 Locking Strategy API

```
1  /*
2   * Tries to acquire an exclusive lock for a process. Fails
        if the lock cannot
3   * be acquired.
4   *
5   * Returns 0 on success, or negative on error.
6   */
7  int get_exclusive_lock(struct client client, uint32_t
      file_id);
8
9  /*
10  * Tries to acquire a shared lock for a process. Fails if
       the lock cannot be
11  * acquired.
12  *
13  * Returns 0 on success, or negative on error.
14  */
15 int get_shared_lock(struct client client, uint32_t file_id
      );
```

```
16
17 /*
18  * Releases a lock, either exclusive or shared. The lock
       released is whatever
19  * kind of lock the process had on the file. Fails if the
       lock is not owned.
20  *
21  * Returns 0 on success, or negative on error.
22  */
23 int release_lock(struct client client, uint32_t file_id);
24
25 /*
26  * Checks whether a process has an exclusive lock.
       Specifying a negative value
27  * for *user_id* or *proc_id* is like a wildcard, and will
        return whether any
28  * *user_id* or *proc_id* has the lock.
29  *
30  * Returns positive if the lock is held, 0 if not, or
       negative on error.
31  */
32 int has_exclusive_lock(struct client client, uint32_t
      file_id);
33
34 /*
35  * Checks whether a process has a shared lock. Specifying
       a negative value for
```

```
36  * *user_id* or *proc_id* is like a wildcard, and will
        return whether any
37  * *user_id* or *proc_id* has the lock.
38  *
39  * Returns positive if the lock is held, 0 if not, or
        negative on error.
40  */
41 int has_shared_lock(struct client client, uint32_t file_id
      );
```

## B.6 IO Manager API

```
1 /*
2  *  Translates a read request from the stripe level to the
       chunk level.
3  *  The correct behavior of this function depends on the
4  *  Distribution and Replication strategies that are in
       place.
5  *
6  *   @return the number of chunks that participated in the
       read
7  */
8 extern "C" uint32_t process_read_stripe (uint32_t
    request_id, uint32_t file_id, char *pathname, uint32_t
    stripe_id, uint32_t stripe_size, uint32_t chunk_size,
    const void *buf, int offset, size_t count);
9
```

```
10
11 /*
12  *  Translates a write request into a series of chunk
       writes and handles
13  *  replication.
14  *  The correct behavior of this function depends on the
15  *  Distribution and Replication strategies that are in
       place.
16  *
17  *   All requests sent to the access module for primary
       storage writes must be
18  *   send with request_id.
19  *   All requests sent to the access module for replica
       writes must bes sent with
20  *   replica_request_id.
21  *
22  *   The number of requests sent to the access module for
       primary storage writes
23  *   must be returned in chunks_written.
24  *   The number of requests sent to the access module for
       replica writes must
25  *   be returned in replica_chunks_written.
26  */
27 extern "C" void process_write_stripe (uint32_t request_id,
       uint32_t replica_request_id, uint32_t *chunks_written,
       uint32_t *replica_chunks_written, uint32_t file_id,
     char *pathname, uint32_t stripe_id, uint32_t
```

```
    stripe_size, uint32_t chunk_size, const void *buf, int
    offset, size_t count);
28
29
30  /*
31   *   Delete all chunks and replicas for a given file.
32   *
33   *   @return the number of chunks that participated in the
         delete
34   */
35  extern "C" uint32_t process_delete_file (uint32_t
    request_id, uint32_t file_id);
36
37  /*
38   * Get information about the storage locations of chunks
       within a file.
39   */
40  extern "C" char * process_file_storage_stat (struct
    file_storage_stat file_info);
41
42  /*
43   *  Set the storage location (node id) for a given chunk
       of a file.
44   *   @return the node id
45   */
46  extern "C" int set_node_id (uint32_t file_id, uint32_t
    stripe_id,
```

```
47                                  uint32_t chunk_num, uint32_t
                                       node_id);
48
49 /*
50  *  Get the storage location (node id) for a given chunk
       of a file.
51  *   @return CHUNK_NOT_FOUND if the chunk hasn't been
       stored <properly>
52  */
53 extern "C" int get_node_id (uint32_t file_id, uint32_t
       stripe_id, uint32_t chunk_num);
54
55 /*
56  *  Set the storage location (node id) for a given replica
        of a
57  *  chunk of a file.
58  */
59 extern "C" int set_replica_node_id (uint32_t file_id,
       uint32_t stripe_id, uint32_t chunk_num, uint32_t
       node_id);
60
61 /*
62  *  Get the storage location (node id) for a given replica
        of a
63  *  chunk of a file.
64  */
```

```
65 extern "C" int get_replica_node_id (uint32_t file_id,
     uint32_t stripe_id, uint32_t chunk_num);
66
67 /*
68  *  Fill in struct decafs_file_stat structure that
     provides information
69  *  about where the chunks live for a specific file.
70  */
71 extern "C" int stat_file_name (char *pathname, struct
     decafs_file_stat *buf);
72 extern "C" int stat_file_id (uint32_t file_id, struct
     decafs_file_stat *buf);
73
74 /*
75  *  Fill in struct decafs_file_stat structure that
     provides information
76  *  about where the stripes live for a specific file.
77  */
78 extern "C" int stat_replica_name (char *pathname, struct
     decafs_file_stat *buf);
79 extern "C" int stat_replica_id (uint32_t file_id, struct
     decafs_file_stat *buf);
80
81 /*
82  *  Ensure that all filedata is written to disk.
83  */
84 extern "C" void sync();
```

## B.7 Distribution Strategy API

```
1 /*
2  *  Determine which node a given chunk from a stripe
     should be sent to.
3  */
4 extern "C" int put_chunk (uint32_t file_id, char *pathname
    , uint32_t stripe_id, uint32_t chunk_num);
```

## B.8 Replication Strategy API

```
1 /*
2  *  Determine which node a given chunk s replica should
     be sent to.
3  */
4 extern "C" int put_replica (uint32_t file_id, char *
   pathname, uint32_t stripe_id, uint32_t chunk_num);
```

## B.9 Access API

```
1 /*
2  *  Read data from a chunk at a specific offset.
3  *  If you are implementing this function:
4  *    If data is being read from an Espresso node, Network
5  *    Layer network_read_chunk() must be called.
6  */
```

```
7  ssize_t process_read_chunk (uint32_t request_id, int fd,
     int file_id,int node_id, int stripe_id, int chunk_num,
     int offset, void* buf, int count);
8
9  /*
10  *  Write data to a chunk at a specific offset.
11  *  If you are implementing this function:
12  *    If data is being written to an Espresso node,
      Network
13  *    Layer network_write_chunk() must be called.
14  */
15 ssize_t process_write_chunk (uint32_t request_id, int fd,
     int file_id, int node_id, int stripe_id, int chunk_num,
      int offset, void *buf, int count);
16
17 /*
18  * Delete a specific chunk from DecaFS.
19  */
20 ssize_t process_delete_chunk (uint32_t request_id, int
     file_id, int node_id, int stripe_id, int chunk_num);
```

```
1  /*
2   * Called during DecaFS statup process. This function
       needs to initiate all
3   *   module-defined startup activities and register custom
       modules with
4   *   Barista Core.
5   */
6  extern "C" void strategy_startup();
7
8  /*
9   *  Register a module to be called with a specific timeout
       ,
10  *    repeatedly throughout DecaFS execution.
11  *
12  * If this function is called MORE THAN ONCE the last
       monitor will be the
13  *   monitor in effect.
14  */
15 extern "C" void register_monitor_module (void (*
     monitor_module)(), struct timeval timeout);
16
17 /*
18  *  Register a function to be called on node failure.
19  *
```

```
20  * If this function is called MORE THAN ONCE the last
       handler will be the
21  *   handler in effect.
22  */
23 extern "C" void register_node_failure_handler (void (*
      failure_handler)(uint32_t node_number));
24
25 /*
26  *  Register a function to be called on node coming online
       .
27  *
28  * If this function is called MORE THAN ONCE the last
       handler will be the
29  *   handler in effect.
30  */
31 extern "C" void register_node_up_handler (void (*
      up_handler)(uint32_t node_number));
32
33 /*
34  * Call a previously registed node failure handler.
35  */
36 extern "C" void run_node_failure_handler (uint32_t
      node_number);
37
38 /*
39  * Call a previously registed node up handler.
40  */
```

```
41 extern "C" void run_node_up_handler (uint32_t node_number)
     ;
```

### B.11 Espresso Storage API

```
1  /*
2   * Reads *count* bytes from the chunk at offset *offset*
       into *buf.
3   * Fails if the chunk doesn't exist, or if the range [
       offset,
4   * offset+count) falls outside the bounds of the chunk.
5   *
6   * Returns the size read, as reported by read(2), or -1 on
        error.
7   */
8  ssize_t read_chunk(int fd, int file_id, int stripe_id, int
       chunk_num, int offset, void *buf, int count);
9
10 /*
11  * Writes *count* bytes from *buf* to the chunk at offset
       *offset*.
12  * Creates a new chunk if it doesn't exist, and resizes
       the chunk if the
13  * range [offset, offset+count) falls outside the exsiting
        bounds of
14  * the chunk.
15  *
```

```
16  * Returns the size written, as reported by write(2), or
       -1 on error.
17  */
18 ssize_t write_chunk(int fd, int file_id, int stripe_id,
      int chunk_num, int offset, void *buf, int count);
19
20 /*
21  * Deletes a chunk, freeing the space it occupied for
       future use. Fails
22  * if the chunk doesn't exist.
23  *
24  * Returns 0 on success, or -1 on error.
25  */
26 int delete_chunk(int fd, int file_id, int stripe_id, int
      chunk_num);
```