PLATFORMS FOR TEACHING DISTRIBUTED COMPUTING CONCEPTS TO UNDERGRADUATE STUDENTS

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

Jeffrey Forrester

March 2015

COMMITTEE MEMBERSHIP

TITLE:	Platforms for Teaching Distributed Computing Concepts to Undergraduate Students
AUTHOR:	Jeffrey Forrester
DATE SUBMITTED:	March 2015
COMMITTEE CHAIR:	Chris Lupo, Ph.D., Associate Professor of Computer Science
COMMITTEE MEMBER:	Alex Dekhtyar, Ph.D., Professor of Computer Science
COMMITTEE MEMBER:	Franz Kurfess, Ph.D., Professor of Computer Science

ABSTRACT

Platforms for Teaching Distributed Computing Concepts to Undergraduate Students

Jeffrey Forrester

Over the last two decades, information technology has been moving towards distributed computing to host their applications and services. These systems can process more data more reliably than their central processing counterparts; however, distributed applications are more complex to design and develop because they require additional properties like replication and fault tolerance to work effectively. These complexities translate to the educational setting, where schools need to invest in additional infrastructure, knowledge, and technologies to teach distributed concepts to students.

This project presents the design and implementation of a complete educational framework for the teaching of distributed computing concepts at Cal Poly. The framework consists of three components: a Raspberry Pi cluster, a custom distributed file system (DecaFS), and a set of labs that can be used to support coursework in a distributed computing class. Each cluster is composed of five networked Raspberry Pi computers. The DecaFS distributed file system runs on the Raspberry Pi cluster. DecaFS provides the base functionality of a distributed file system with a design that allows for easy modification of sections of the implementation. The lab exercises focus on important distributed computing concepts that represent a variety of problems encountered in distributed systems including distribution, replication, fault tolerance, recovery, rebalancing, and efficiency. Isolation of the lab related modules allows students to focus on the learning objectives of the labs without needing to set up network and file system infrastructure to support the distributed aspects.

The complexities of teaching distributed computing concepts in a classroom set-

ting at Cal Poly have been addressed with this project's framework. The solution overcomes key educational challenges as it is portable, modular, scalable and affordable. The framework provides the ability to offer courses in distributed computing to better prepare students for the challenges presented in industry today. Through the use of a modular distributed file system and computing cluster that were created for this project, students are able to solve complex distributed problems, in the form of labs, in an isolated environment that is conducive to quarter long learning objectives.This work is a major step to bringing distributed computing into the classrooms at Cal Poly and classes are currently being designed around this curriculum. Cal Poly can evolve the framework to keep pace with the ever advancing information technology world so that it may continue to serve the needs of the faculty and students of Cal Poly.

ACKNOWLEDGMENTS

Thanks to:

- Chris Lupo
- Alex Dekhtyar
- Halli Meth
- Peter Faiman

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

1.1 Undergraduate Distributed Computing Education

Many software professionals work on large-scale distributed computing applications. These "internet-scaled" applications run on large distributed systems and process enormous amounts of data. To support the development of these systems, software engineers use highly-parallel computing practices to solve issues that occur in these types of systems [16].

At the undergraduate level, computer science students work on labs and projects that generally run on their local computer. Universities attempt to bring the distributed computing paradigm into their classrooms, yet they encounter many barriers including infrastructure cost, prerequisite knowledge, and rapidly changing industry technologies that make the introduction of these classes extremely difficult.

If universities overcome these barriers, they can offer courses in distributed computing to "better prepare [their] students for the challenges presented by highly parallel computing" [16]. To prepare computer science students for industry, courses in distributed computing often cover the following topics:

- distributed programming models
- concurrency and synchronization
- distributed database systems

- network topologies
- fault tolerance, reliability, and availability
- testing methodologies
- security
- cloud computing
- peer-to-peer

1.2 Our Contributions

This thesis presents a complete educational framework for the teaching of distributed computing programming at Cal Poly. Through the use of a modular distributed file system and computing cluster that were created for this project, students are able to solve complex distributed problems, in the form of labs, in an isolated environment that is conducive to quarter long learning objectives.

The specific contributions of this thesis are:

1. Creation of a Raspberry Pi cluster that can be used in a hands-on classroom setting.
2. Creation of Distributed Educational Component Adaptable File System (DecaFS)
3. Outline of distributed computing classroom labs to extend the base functionality of DecaFS.

The distributed cluster of Raspberry Pis runs DecaFS, a modular distributed file system, which is described in detail in my colleague's work [10]. The labs described

later in this paper modify isolated portions of DecaFS to achieve new or different behaviors related to the fundamental properties of distributed systems.

1.3 Outline of Chapters

CHAPTER 2 provides detailed background information on Distributed File Systems. Related Work is documented in CHAPTER 3. CHAPTER 4 describes the Raspberry Pi cluster used to host DecaFS. An overview of DecaFS is presented in CHAPTER 5 followed by a description of suggested labs in CHAPTER 6. CHAPTER 7 covers the testing and validation of the Raspberry Pi clusters, DecaFS, and Labs presented in the previous chapters. Conclusions and Future Work are presented in CHAPTER 8 and CHAPTER 9.

CHAPTER 2

Background

2.1 Distributed Systems

For the purposes of this paper, a loose definition of distributed systems is used: "A distributed system is a collection of independent computers that appears to its users as a single coherent system [21]." This definition highlights two important aspects of distributed systems. First, distributed systems are made up of many computers that act together; and second, that this group of computers is indistinguishable from a single computer to its' clients.

Distributed systems are used in all sectors of modern computing and are commonly used to host software applications and services today. The distributed paradigm rose to prominence due to the many advantages they offer over centralized systems. Distributed systems make it easy to integrate applications and they scale well with the underlying network [21].

These advantages do not come without a cost. Applications written on top of distributed systems must deal with issues that do not exist on centralized systems, such as network latency, inconsistent clocks, and node failure. The added complexity increases the difficulty of design, implementation, and maintenance of these systems as they must be taken into account by software developers. These difficulties are multiplied because designers often make assumptions about these systems that are known as the fallacies of distributed computing [20]:

1. The network is reliable
2. Latency is zero
3. Bandwidth is infinite
4. The network is secure
5. Topology doesn't change
6. There is one administrator
7. Transport cost is zero
8. The network is homogeneous

2.1.1 Fundamental Properties

While designers often overlook network properties that affect how a distributed system operates, the following desirable properties are usually designed into such a system.

2.1.1.1 Replication

Replication is a desired characteristic of distributed systems because it helps systems be fault tolerant and allows them to scale their throughput. In distributed computing, replication refers to the duplication of data, software, or tasks across more than one physical or virtual machine. Replication is an important aspect of the foundation of distributed computing.

2.1.1.2 Fault Tolerance

Fault tolerance is the ability of a service to continue running at full or partial capability in the event of node failure. Node failure is common in distributed systems and is

often accounted for in the design of distributed applications like Google File System that is discussed in Section 3.1. Node failure has many causes which include: application failure, power loss, and network connectivity loss. To achieve fault tolerance, distributed systems replicate resources across multiple nodes and redirect usage of the replicated resources to other nodes. In some cases, combinations of certain nodes failing can take down access to small parts of the service (data or functionality), but will only take down the entire system in extreme cases.

2.1.1.3 Availability

Highly available systems can achieve over 99.9% availability by being fault tolerant and quickly resolving critical system failures [22]. Availability is measured by the percentage of time that a service is responsive to clients within the bounds of the program's service level agreement (SLA). What percentage of time is a service available to perform its defined function? Availability is desirable because down-time of commercial products impacts the revenue of the product [22].

2.1.1.4 Scalability

Scalability is the ability of a system to easily adjust to increases (or decreases) in demand. Scalability can be measured by the throughput and quality of services of a system. Can the system be scaled to serve more clients from many regions at the same time? A scalable system allows products to scale the throughput of their system as demanded by its clients. As it serves more clients, it can scale to serve all of them with the same quality of service as it did when there was less throughput or fewer clients. Like fault tolerance and availability, scalability is also achieved using replication.

2.1.1.5 Transparency

Transparency exists in many aspects of distributed systems. It is most often thought of as the appearance of the distributed system as a single coherent system; however, table 2.1 contains many types of transparency that can be built into distributed systems:

Transparency	**Description**
Access	Hide differences in data representation and how a resource is accessed
Location	Hide where a resource is located
Migration	Hide that a resource may move to another location
Relocation	Hide that a resource may be moved to another location while in use
Replication	Hide that a resource is replicated
Concurrency	Hide that a resource may be shared by several competitive users
Failure	Hide the failure and recovery of a resource

Table 2.1: Types of Transparency [21]

2.2 Distributed File Systems

Distributed applications, written on top of distributed systems, can take advantage of the desireable properties to offer additional functionality over their centralized counter parts. One such distributed application, a distributed file system (DFS), can offer users the data storage and security of a traditional file system, while also providing the users properties like fault tolerance, scalability, and availability that a traditional file system cannot provide.

Industry distributed file systems are designed to meet the needs of their designers or their target customers. One such DFS, Google File System (GFS), is designed to meet the data processing needs of Google with goals of: performance, scalability, reliability, and availability [7]. While these systems share similar goals with DecaFS, they do not provide the ability to alter their implementation to substitute other algorithms to achieve their goals. This is the primary reason we decided to implement our own distributed file system that provides the capability to easily modify sections of the implementation and can run on extremely limited hardware such as a Raspberry Pi.

2.3 Raspberry Pi

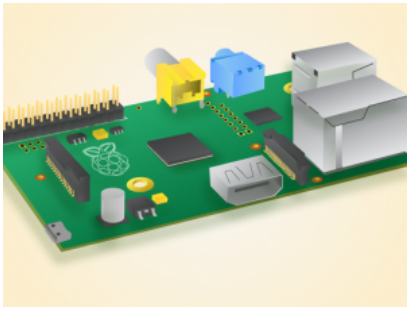

Figure 2.1: Model B Raspberry Pi [2]

A Raspberry Pi is a "low cost, credit-card sized computer" [4] capable of running operating systems compiled for the ARM architecture such as Raspbian and Arch-Linux. The Model B Raspberry Pi computers used in this project have the following specifications [1, 3]:

- Full size SD card
- HDMI output port

- Composite video output
- Two (2) USB ports
- 26 pin expansion header exposing GPIO, I2C etc
- 3.5mm audio jack
- Camera interface port (CSI-2)
- LCD display interface port (DSI)
- One microUSB power connector for powering the device
- 512 MB of SDRAM
- One ethernet port
- 750 MHz ARMv6 CPU

A basic Raspberry Pi system can be built with a very small budget of around $50 USD. A simple system includes a Raspberry Pi ($35 USD), a SD card, and an ethernet cable. A free open source operating system, such as Raspbian or Arch-Linux, can be installed to keep costs minimal. The ethernet cable connects the Raspberry Pi to a router if network access is desired. Additionally, a USB 2.0 keyboard and an HDMI monitor can be connected to allow the use of the desktop interface provided by the chosen operating system.

CHAPTER 3

Related Work

The following chapter our background research into distributed file systems, and distributed systems designed for educational use in an attempt to bring distributed computing into the classroom at Cal Poly.

3.1 Distributed File Systems

Distributed file systems (DFS) are used throughout industry to support large-scale distributed computing applications that process large volumes of data. Unlike other distributed file systems, DecaFS is designed modularly to support educational requirements unique to our project.

3.1.1 Lustre

Lustre is a cluster file system that aims to be massively scalable and has been tested with 50,000+ clients, a billion files, and 55 petabytes (PB) of total storage [11]. It is an open source POSIX compliant file system that focuses on high availability, heterogeneous network support, scalability, and data security [11]. Lustre is the "most widely used file system by the world's Top 500 HPC [High Performance Computing] sites [18]."

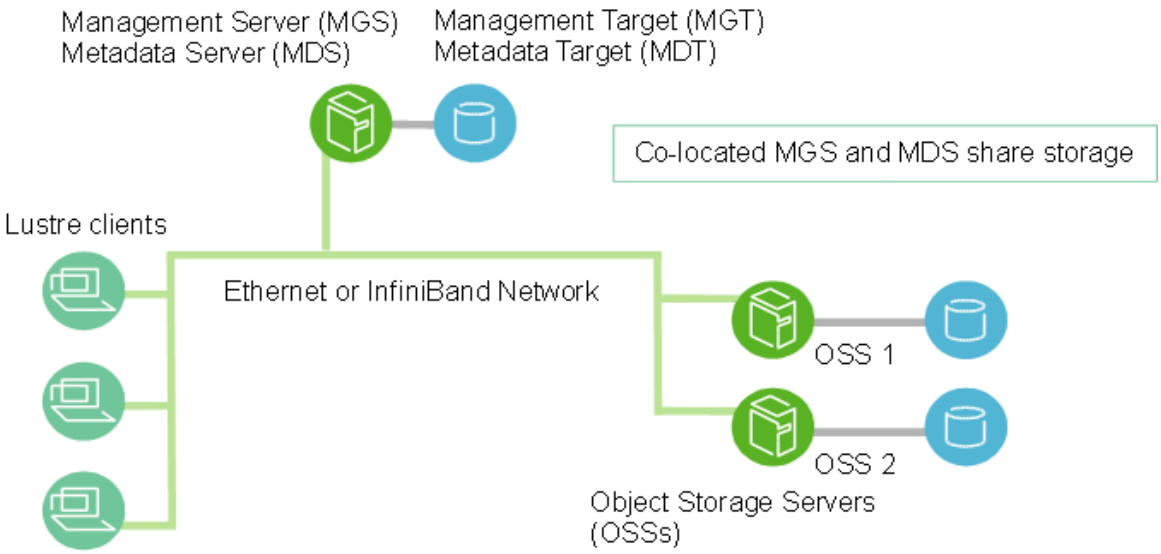

Figure 3.1: Lustre Architecture [11]

A typical Lustre cluster is composed of: a Management Server (MGS), a Metadata Server (MDS), Object Storage Servers (OSSs), and many clients. The MDS stores the metadata of the files including filenames, directories, and file permissions and makes this information available to clients. The OSSs are the file access providers, and typically store filedata on two to eight disks that offer up to 16 terabytes (TB) of total storage space. Each of the servers has corresponding targets with physical disks for data storage..

Lustre is not suited to this project for many reasons. While it can run on commodity hardware, it benefits from specialized enterprise hardware like storage arrays and storage area networks that far exceed the budget of this project. Additionally, Lustre does not replicate data at the file system software layer and relies on failover techniques that would not be supportable at Cal Poly.

3.1.2 GFS

Unlike Lustre, Google File System (GFS) is designed to run on commodity hardware. It is a "scalable distributed file system" that meets Google's data processing needs with goals of performance, scalability, reliability, and availability [7]. While having the same goals as many distributed file systems, GFS deviates from previous assumptions made by these types of file systems based on their needs. GFS is designed to run on commodity hardware, support large file sizes, support concurrent appending to files, and prioritizes high bandwidth over low latency [7].

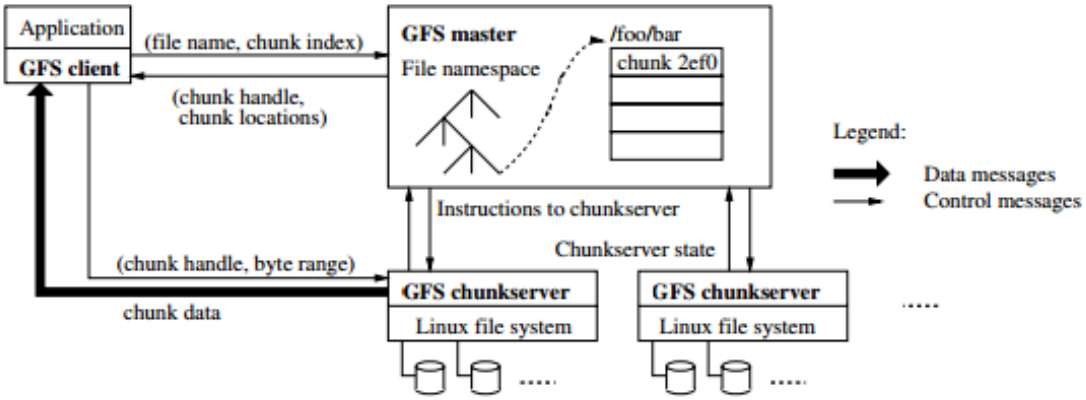

Figure 3.2: GFS Architecture [7]

A GFS cluster is composed of a single *master* node and many *chunkservers*. The *master* node is responsible for tracking the metadata of the system which includes: file and chunk namespaces, mapping from files to chunks, and locations of chunk replicas. In a GFS instance, there is a single *master*, which allows it to make intelligent chunk storage decisions based on its global knowledge of the system. This metadata is stored in-memory to keep *master* operations fast. To maintain fault tolerance of the system, operation logs track metadata changes and the operation logs are stored to disk on

the *master* and replicated on some of the *chunkservers* of the system. *Chunkservers* are used to store the file chunks as determined by the *master*. These data transfer operations are not routed through the master as clients transfer this data directly to the chunkservers. The GFS design and implementation also allow for atomic file creation and file appending that they believed was beneficial to their data intensive applications.

The design and implementation of GFS fit Google's requirements, but like Lustre, GFS is not appropriate for the needs of this project. GFS is designed for large data files, large sequential reads, and concurrent appending to data files. This project's goal is to support classroom projects that do not have the same data needs or hardware infrastructure.

3.1.3 HDFS

Hadoop Distributed File System (HDFS) is a "distributed file system designed to run on commodity hardware" [5]. The goals and assumptions of HDFS are almost identical to those of GFS: provide high-throughput streaming data access, treat hardware failure as the norm, design for large data sets, and prioritize high throughput over low latency. Additionally, HDFS assumes that "moving computation is cheaper than moving data" [5], and tries to move computations to the data whenever possible.

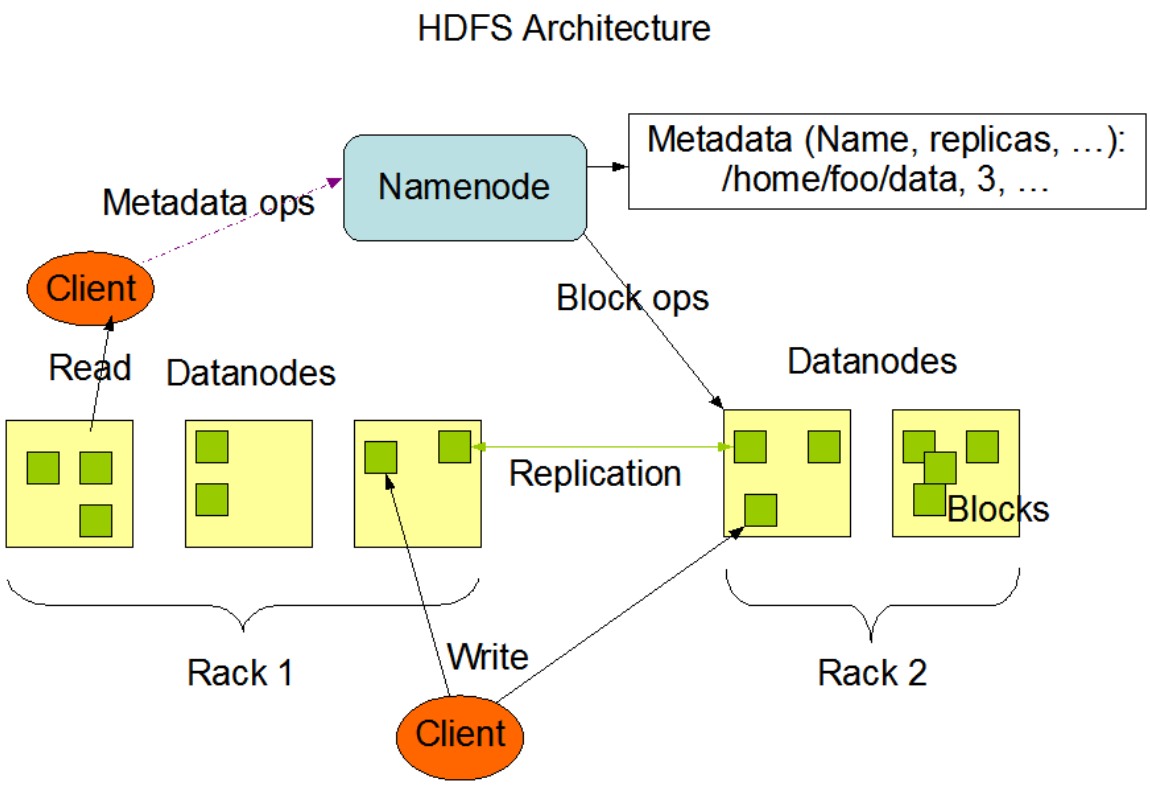

Figure 3.3: HDFS Architecture [5]

The HDFS architecture uses two types of nodes: a single namenode, and multiple datanodes. Like the GFS master node and the Lustre Metadata server, the HDFS namenode keeps track of file metadata (filenames and namespaces) in the system, and handles data access for clients. The namenode stores the datanode locations of blocks and directs clients to the correct datanodes for their data operations. Datanodes store blocks of files and service client data operations like read and write. To provide fault tolerance, blocks are replicated across multiple datanodes based on a replication factor (how many times a block should be replicated). The namenode uses a transaction log that records updates to the system and is used to recreate the persistent metadata stored on the namenode in the case of its failure.

HDFS is open source and was considered for use in this project, but we were unable

to compile and run HDFS on Raspberry Pi computers. A member of the Raspberry Pi community successfully installed HDFS and Hadoop on a small Raspberry Pi cluster [12]. Their cluster was extremely slow (1.2 MB/s write speed). They attributed the bottleneck to either the Java Virtual Machine (JVM) or Network Interface Card. One proposed solution was to implement a hadoop-like file system in C/C++ to eliminate the suspected bottleneck of the JVM. This is the solution we decided upon when we decided to create DecaFS.

3.2 Distributed Computing Education: Curriculum and Platforms

In [8], Hoganson addresses the needs of a well rounded computer science education that prepares students for a globally competitive job market. Integral to this curriculum are the core technologies behind today's applications: networking, distributed computing, human-computer interface, and security [8]. While including distributed computing at both the undergraduate and graduate computer science levels, a model for teaching the distributed concepts is not addressed.

3.2.1 Seattle

A platform for educational distributed systems is provided in [6]. This platform, Seattle, runs on a set computers and supports "cloud computing, distributed systems, grid-computing, peer-to-peer networking, distributed systems, and networking" [6]. Compute time is donated by contributors from unused resources of machines. The platform runs on a set of heterogeneous machines and *Seattle* provides a programming API for non-portable network communication and file operations that allow students to write portable programs. A manager portion of *Seattle* performs monitoring of the programs to protect donated resources from malicious and poorly-written code.

3.2.2 Beowulf Cluster

Kiepert describes how he built another educational platform, a Beowulf Cluster, out of a group of 33 Raspberry Pi computers [9]. The author built the system by connecting the Raspberry Pi computers over a switch and setting up a network file system available to all of the units. The Raspberry Pis are housed in a custom built enclosure that uses fans for cooling and contains a power supply to support the power requirements of the Raspberry Pis. Overall, the case mimics a desktop computer tower.

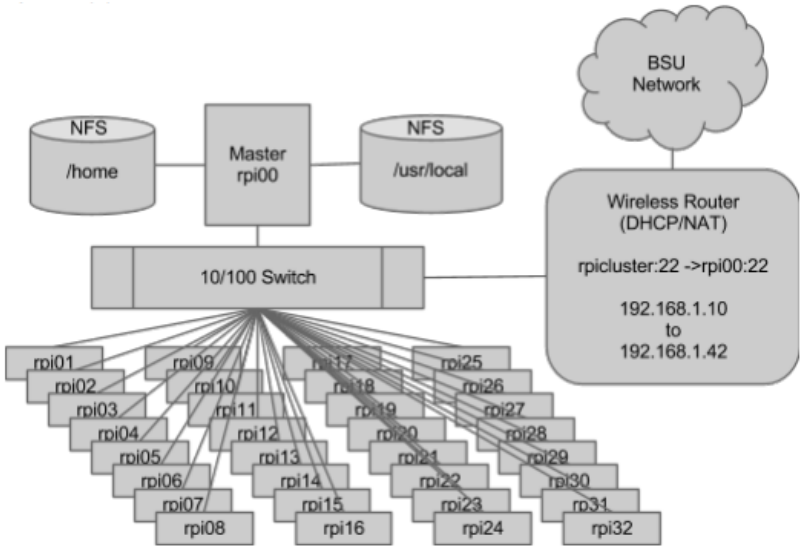

Figure 3.4: Network Architecture for Raspberry Pi Cluster [9]

The author concludes that the cluster mimics real distributed systems and is great for testing distributed software as it scales well with additional nodes. Downsides of the cluster are that each node has a limited amount of RAM (just 512 MB) so it cannot support multiple clients simultaneously, and that the use of the ARM architecture requires custom compilation of some programs. We found these traits desireable, and ideal for our target classroom environment leading to our decision to use Raspberry Pi computers in our own clusters.

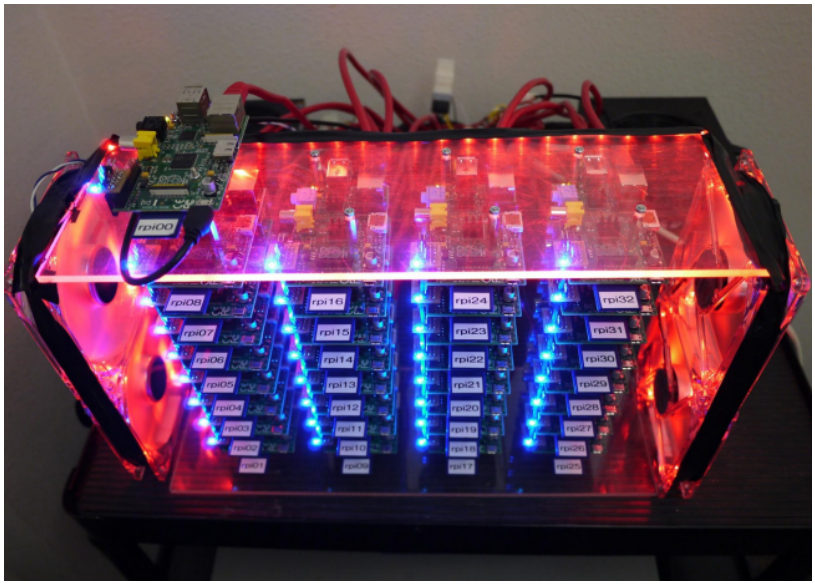

Figure 3.5: Beowulf Raspberry Pi Cluster [9]

CHAPTER 4

Raspberry Pi Cluster

When we began requirements and design for this project, our overall objective was to align with Cal Poly's hands-on learning approach by providing an environment where students could learn by doing.

4.1 Goals & Requirements

Our primary goal for the distributed computing platform was to construct a system where students could "play" to learn more about distributed computing. This included building physical clusters of computers that could support a class size of approximately 30 students. Breaking classes into groups of three to four students each equated to a minimum of 8 clusters. Each cluster would need to include enough nodes to create a viable network or networks to support the goals of the distributed computing labs as described in CHAPTER 6. The hardware setup to support these labs required a minimum of one master with two pairs of slave nodes, totaling a minimum of five nodes per cluster. As students would be working directly with the clusters, we also needed something that was physically small in size and portable from classroom to classroom. The clusters would need to be modular in order for students to assemble or disassemble as needed per requirements for their labs. An educational grant was obtained from CP-Connect for the clusters, so we needed the clusters to be affordable within the budget of the grant. This applies to both hardware and software, so free open source software is required. Finally, it was also desirable for

the clusters to be scalable with options to add nodes to a cluster or create additional clusters following the same architecture.

In summary, requirements for the platform were:

1. 8 clusters with minimum of 5 nodes per cluster
2. Physically small enough to be portable classroom to classroom
3. Modular in order to assemble / disassemble as desired
4. Affordable within the budget of CP-Connect grant (approximately $4000)
5. Scalable in quantity of nodes and additional clusters
6. Use of free open source software (operating system and tools)

4.2 Design

To fulfill the size, portability, and budget requirements, we needed to build the clusters with computers that were also very small, portable, and affordable. This restricted us to a Single Board Computer (SBC) as this was the only option we believed could meet our requirements for the clusters. We selected the Raspberry Pi because it offers a solution for size and cost with a level of performance that we felt was acceptable. Another benefit of using Raspberry Pis as the computing platforms for the cluster is that they only have a 750 MHz ARM processor. This means that they can be easily pushed to their limits and can simulate industry data problems with a fraction of the data.

For the setup of each individual Raspberry Pi, we wanted to provide students with an environment with all of the tools and libraries they might need along the way. With ease of development and use in mind, we selected the Raspbian operating

system because it comes default with a desktop environment and many other packages. By using this Linux distribution, students have the option to plug a monitor and keyboard into a Raspberry Pi node to test and develop on the nodes. One downside of using the Raspbian images is that there are unnecessary libraries and programs installed on the operating system taking up some of the limited amount of processing power and disk space. We considered an alternative, ArchLinux, that is a minimalist operating system which does not come with a desktop environment or other useful libraries. Instead, you have to install all of the tools you need separately. To avoid students having to perform system administration roles, we felt Raspbian was a better solution as it provides a larger set of base functionality.

In addition to the operating system selection, we also wanted to provide all necessary tools and libraries for the students. Packages we deemed important included: the vim text editor, the GNU C compiler, the GNU C++ compiler, and the OpenMPI library. Vim and the GNU compilers provide students with tools to develop and compile code on the computers. OpenMPI is a Message Passing Interface library that is useful for creating distributed computing programs and is used in the MapReduce lab discussed in CHAPTER 6.

4.3 Building the Clusters

Building the physical clusters was straightforward as there were not many parts to setup and configure. The required hardware components were the Raspberry Pis, the wireless routers, and the USB hubs. Additional hardware included housings, data storage, and networking components.

No.	Item	Quantity	Price Per Item	Total
1	Raspberry Pi Model B board	40	$35	$1400
2	Plastic Enclosure for #1	40	$10	$400
3	Micro USB cord	40	$10	$400
4	SD Card (8 GB)	80	$10	$800
5	5 (7)-port USB 2.0 hub	8	$30	$240
6	USP port splitter	40	$5	$200
7	USB Wi-Fi dongle	40	$12	$480
8	Wi-Fi N600 Router	2	$80	$160
	Total			$3920

Table 4.1: Bill of Materials for 8 Clusters at the time of purchase (FALL 2013)

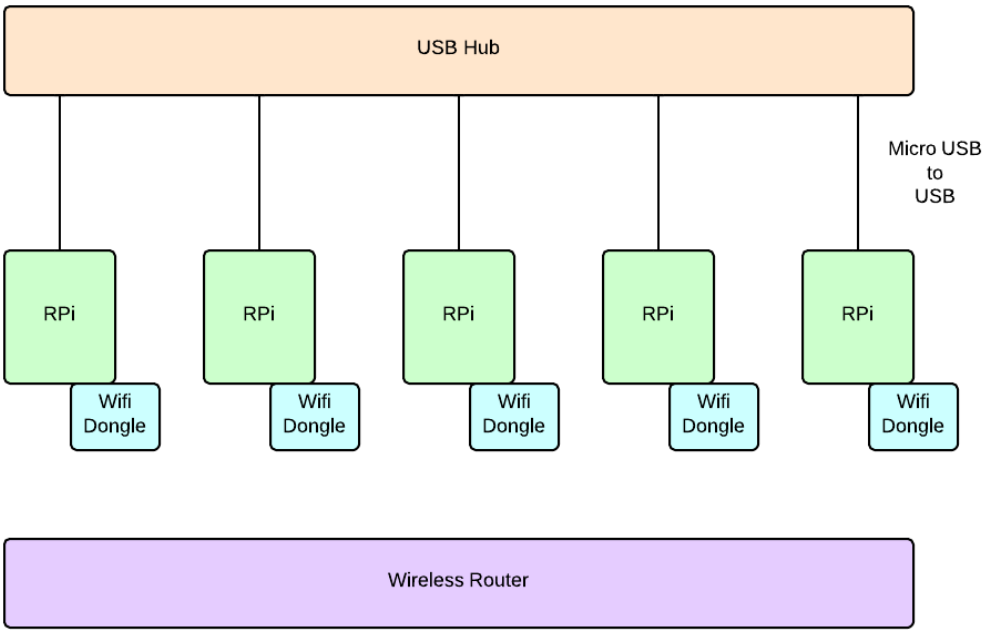

Figure 4.1: Raspberry Pi Cluster Setup.

Each cluster was built by connecting five Raspberry Pi computers to a single USB hub, which provides power. Each Raspberry Pi was connected to a wireless router via wireless dongles, which provides wireless network connectivity to the cluster. Each Raspberry Pi was assigned two SD cards that provide bootable replaceable storage.This allows shared use by multiple groups without interference because multiple operating systems and data can be maintained. The bill of materials totaled $3920, an average of $490 per cluster. If desired, clusters can be enhanced to use switches and ethernet cables for a faster, more reliable network connection. The clusters could then share a switch and have the switch plugged into the wireless routers for student access, but would be slightly less portable as they would require additional hardware.

To make installation of the operating system and required libraries simple for students and future users of the clusters, we created an image of the Raspbian operating system with vim, GNU gcc, GNU g++, and OpenMPI installed. This image can be hosted on a professor's webpage. To prepare a cluster for a class, the image can be downloaded and installed on each of the SD cards of the cluster. This is the starting point where the cluster is ready to be used for distributed application processing.

4.4 Conclusion

In summary, we were able to build eight low cost distributed computing clusters using Raspberry Pi computers with each cluster fitting in the size of a shoebox. The size is advantageous, because students can take the cluster or individual Raspberry Pi home and connect them to their home router for development and testing outside of the classroom. These clusters support the proposed classroom labs from this project and are scalable for larger projects in the future.

CHAPTER 5

DecaFS

DecaFS is the distributed file system that we designed, developed, and deployed on the Raspberry Pi clusters described in CHAPTER 4. An overview of DecaFS is provided in this Chapter. For an in-depth description of the design and implementation of the distributed file system see my colleague's [10].

5.1 Terms and Definitions

1. *Node*: A machine running a component of DecaFS
2. *Barista Node*: A machine running the *master* node of DecaFS, one per instance of DecaFS, also referred to as the *Barista*
3. *Espresso Node*: A machine running a *slave* node of DecaFS, at least one per DecaFS instance
4. *Module*: An isolated portion of code that lives in its own library, can be compiled independently, and usually contains only a few functions
5. *Stripe*: A logical piece of a file that will be distributed across Espresso Nodes
6. *Chunk*: A portion of a stripe, a chunk is the data written to an Espresso Node
7. *Primary Chunk*: A chunk marked as the primary (unmodified) data
8. *Replica Chunk*: A non-Primary marked chunk used to achieve fault tolerance,

can be stored unmodified or as a parity of other chunks in the stripe, generally not stored on the same Espresso Node as the primary chunk

9. *DecaFS Client*: Any user running a process on a node in DecaFS

5.2 Requirements

The overall goal of DecaFS is to support the teaching of distributed computing concepts to students at Cal Poly. To fit as many foundational distributed computing concepts into a quarter as possible, a design that allows students to quickly modify portions of the system is a must. In addition, we want students to be able to develop techniques to perform common distributed tasks that are deemed core concepts including distribution, replication, and recovery.

5.2.1 Labs

At Cal Poly, computer science classes include labs where programming projects are assigned to reinforce and provide hands on practice for concepts and topics covered in class lectures. In order to make each lab focused on specific learning goals, we wanted to create a system which students could modify one module at a time. The following labs were considered in the design of the file system and are described in CHAPTER 6:

1. Distribution Strategies
2. Replication Strategies
3. Storage and Recovery Performance
4. Caching

5. Adaptive Data Migration

6. MapReduce

5.3 DecaFS Design

DecaFS is designed modularly so that students can modify small sections of the system at a time to align with lab assignments. We separated lab related code into modules isolated from the modules that control the core functionality of the file system. Network communication, reads / writes, and persistent metadata storage related to the file system aspect of DecaFS are also separated from the lab related modules so that students can focus on implementing distributed computing concepts instead of worrying about lower-level details of the file system. DecaFS is split into three layers: Barista Layer, Network Layer, and Espresso Layer. The Barista Layer runs on the master node of the system and contains the code that distributes the file system. The Network Layer deals with the communication between nodes, and the Espresso Layer is responsible for the storage and retrieval of file data in the system.

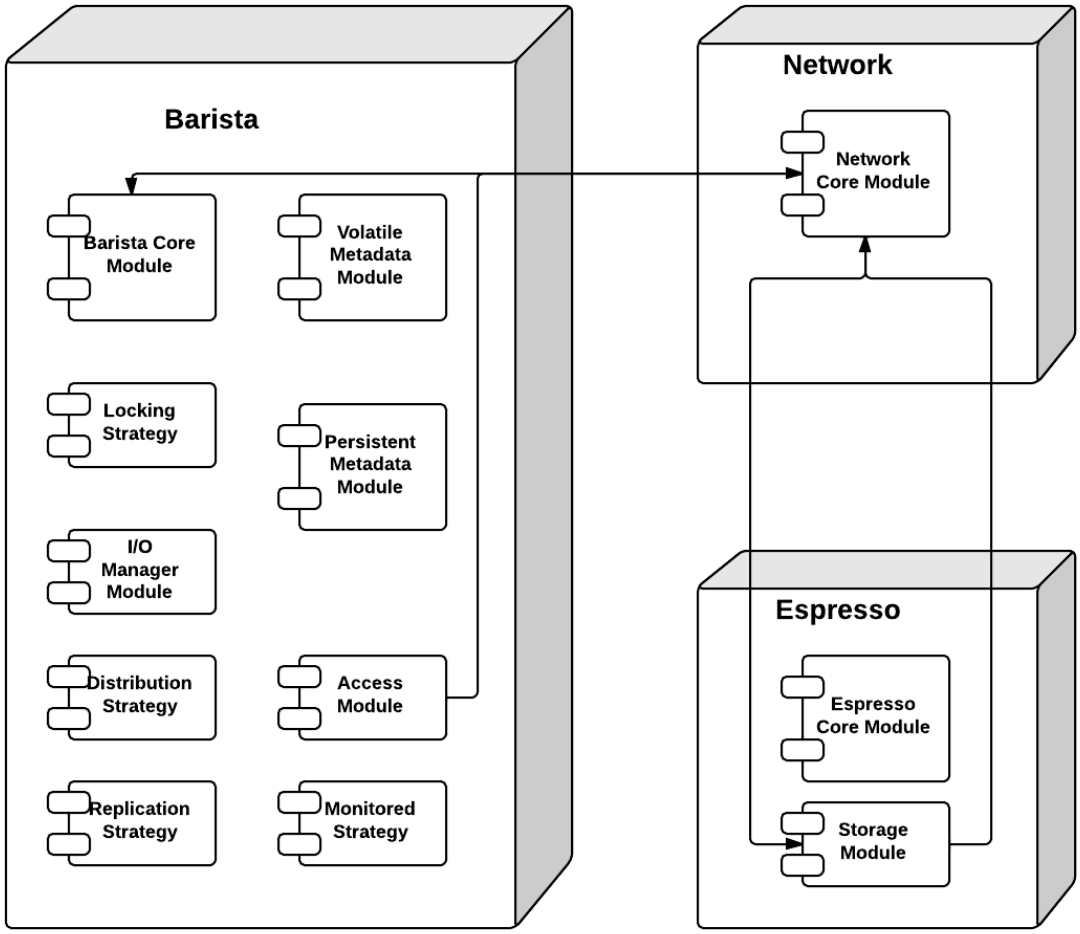

Figure 5.1: Architecture for DecaFS.

5.3.1 Barista Layer

The Barista Layer handles the metadata of the file system and is the client's point of contact into the file system. The modules of the Barista Layer are coordinated by the Barista Core Module and the details of how the file system behaves are defined in the other Barista modules.

5.3.1.1 Barista Core Module

The Barista Core Module coordinates all client requests that come into the file system. It controls the data flow through all other modules within the Barista Layer and controls the startup and shutdown functionality of the file system. It coordinates the Espresso Nodes and monitors their states.

5.3.1.2 Volatile Metadata Module

The Volatile Metadata Module tracks details of the file system including a list of open file descriptors that are mapped to corresponding file_id, client_id, and lock_status.

5.3.1.3 Persistent Metadata Module

The Persistent Metadata Module tracks and stores non-recoverable details of the file system. This information includes file metadata such as filename, file_id, filesize, and chunk storage locations.

5.3.1.4 Locking Strategy Module

The Locking Strategy Module controls access to the files in the file system to prevent the corruption of files during concurrent access by multiple clients. DecaFS uses a simple locking mechanism that does not allow shared access to files. A client can have one of two types of locks: exclusive or shared. Exclusive locks are used to write to a file and can only be held by one client process. In order to read from a file, a client obtains shared locks which can be held by more than one client process.

5.3.1.5 I/O Manager Module

The IO Manager Module is responsible for converting data operations from the stripe level to the chunk level. When it receives requests from the Barista Core Module, it separates the requests into the relevant chunks and requests the chunks from their corresponding Espresso Nodes. The Espresso Node location of the chunks are determined by the Distribution and Replication Strategy Modules.

5.3.1.6 Distribution Strategy Module

The Distribution Strategy Module determines how chunks are distributed across Espresso Nodes. The distribution strategy can use file_id, filename, stripe_id, and chunk_num to determine where to send a chunk.

5.3.1.7 Replication Strategy Module

The Replication Strategy Module determines which Espresso Node replica chunks are stored. The replication strategy can use file_id, filename, stripe_id, chunk_num, and node_id to determine where to send a chunk.

5.3.1.8 Access Module

The Access Module is the layer that abstracts the Network Layer from the rest of the Barista Layer. At this level, read, write, and delete operations are performed per chunk.

5.3.1.9 Monitored Strategy Module

The Monitored Strategy Module allows for the handling of node failures and restarts. When either of these events occur, this module is notified and can handle data rebalancing or data synchronization as needed.

5.3.2 Network Layer

The Network Layer facilitates the communication between nodes.

5.3.2.1 Network Core Module

The Network Core Module is split into two portions: Barista server, and Espresso client. The Barista server listens for connections from Espresso clients and notifies the Barista Core Module when Espresso Nodes start, fail, or restart. The client and server are the communication line between the Barista Core Module and the Espresso Core Module and all data transfer and job status messages are passed through this module.

5.3.3 Espresso Layer

The Espresso Layer is responsible for storing file chunks to disk. An instance of the Espresso Layer runs on all Espresso Nodes, and each has its own metadata to keep track of the chunks stored on its node.

5.3.3.1 Espresso Core Module

The Espresso Core Module tracks the persistent metadata of the chunks stored on the Espresso Node. This information relates file_id, stripe_id, and chunk_id to the

location on disk where the chunk is physically written.

5.3.3.2 Storage Module

The Storage Module takes care of reading, writing, and deleting chunks from and to the disks of Espresso Nodes. It is also responsible for disk space management.

5.4 Implementation

In our implementation of DecaFS, we provide base functionality of a distributed file system implemented with striping and mirroring (RAID 10). Given four Espresso Nodes per design, Primary Chunks are stored on nodes 1 and 3, while the corresponding Replica Chunks are stored on nodes 2 and 4. Our Monitored Strategy Module is left empty and does not perform rebalancing of data when a node comes online or goes offline. Implementation of this module is left up to the students and is required in some labs. In addition, more sophisticated methods for distribution, replication, and rebalancing are left to students to implement and are described in CHAPTER 6.

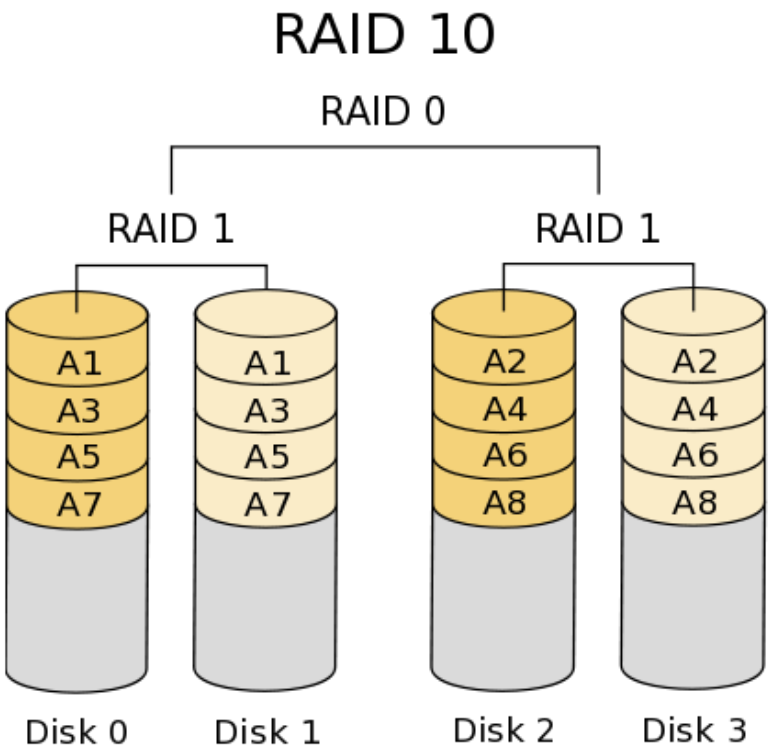

Figure 5.2: RAID 10 Chunk Distribution [17]

5.4.1 Validation Tools

While testing the system and layers of DecaFS, we found it useful to create a file stat function specific to DecaFS to allow users to see where data is stored in the system. This custom informational function, *decafs_file_stat()*, looks up a file name in the system and returns a JSON formatted string with information on how the file is stored in the system.

```
{
  "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
        }
      ]
    }
  ]
}
```


Figure 5.3: *decafs_file_stat() sample output*

This function can be used by testers of the system as an intermediate testing stage to see if the system is storing data as expected. One such intermediate test would be a simple validation to ensure that file chunks are not being stored and replicated on the same node. While this does not guarantee that the file system is fault tolerant, it is a requirement for a system that wants to be fault tolerant. Another verification that could be performed with this function is the differencing of the current result to past results of the storage statistic function. This could be used to test student implementations against the instructor's oracle system. This function has many uses outside the scope of this section that are addressed as possible future work that could be done with DecaFS in CHAPTER 9. This function is the primary tool used to validate student's lab solutions.

CHAPTER 6

Labs

When creating DecaFS and the Raspberry Pi platform, we wanted to make sure we could support a set of labs representative of common distributed computing problems such as: availability, replication, and scalability. Additionally, we wanted students to be able to begin writing distributed algorithms without having to set up any infrastructure (hardware or software). With this platform and the DecaFS distributed file system, students should be able to start writing distributed code right away. Each lab takes advantage of the Raspberry Pi cluster, but does not always use the DecaFS code base. Lab topics we specifically support are:

1. Distribution and Replication
2. Caching
3. Storage and Recovery Performance
4. Adaptive Data Migration
5. RAID 4
6. MapReduce

6.1 Distribution and Replication

For this lab, students will implement a paired distribution and replication strategy that provides one-node fault tolerance within the DecaFS system. The algorithm

functionality for this lab sits in the Distribution and Replication Modules and is designed to introduce students to the Raspberry Pi Cluster and DecaFS.

Learning Objective: To familiarize students with DecaFS and have them start programming core distributed computing concepts

6.1.1 Student Implementation

This lab will be implemented in the Distribution and Replication Modules and requires students to setup an algorithm that stripes and mirrors the data chunks stored on the Espresso Nodes in the DecaFS instance. This lab has students implement RAID 10 to provide one-node fault-tolerance. To do this, students need to implement two functions: `put_chunk`, and `put_replica`.

```
int put_chunk(uint32_t file_id, char *pathname, uint32_t
   stripe_id, uint32_t chunk_num);
int put_replica(uint32_t file_id, char *pathname, uint32_t
    stripe_id, uint32_t chunk_num);
```

Figure 6.1: Distribution and Replication Lab Functions

In both functions, students have *file_id*, *pathname* *filename*, *stripe_id*, and *chunk_num* to determine where to send the chunk. If desired, additional data structures can be used to track where previous chunks have been stored. The *put_chunk* function returns the Espresso Node id to store the primary chunk and the *put_replica* function returns the Espresso Node id to store the mirrored chunk. Once these functions determine and return the node to store the chunk, the system automatically contacts the Persistent Metadata Module to store the metadata information and the Barista Core Module writes the data to the corresponding nodes using the Access Module.

6.1.2 Instructor Evaluation

There are multiple ways to evaluate the accuracy of the implementation of this lab. The simplest method is to use the *decafs_file_stat* function described in Section 5.4.1 that returns a JSON string description of how primary and replica chunks are stored on the Espresso Nodes. To use this method, the student's modified DecaFS implementation should be compiled and started on a Raspberry Pi Cluster. Next step is to write a few files to the DecaFS instance, then call *decafs_file_stat* and verify that the results it returns show that no primary and replica chunks of the same chunk are stored on the same Espresso Node. More in-depth verification of the *stat* results can be performed if the instructor decides to enforce a strict implementation of RAID 10 or another fault-tolerant model.

6.2 Caching

The goal of this lab is for students to explore how network latencies can affect the responsiveness of distributed systems. Can we cache frequently read data to increase read speeds and reduce the number of expensive network requests that are called? How do different caching policies affect hit rates of cached data and what policies work best in distributed file systems?

Learning Objective: To have students explore how network latency affects responsiveness of DecaFS and to implement a simple caching mechanism to increase the read speed of frequently read chunks

6.2.1 Student Implementation

In this lab, students will be isolated to the Access Module where they need to implement three functions: *process_read_chunk*, *process_write_chunk*, and *process_delete_chunk*.

```
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) {


  return network_read_chunk (request_id, fd, file_id,
     node_id, stripe_id, chunk_num, offset, count);
}

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) {

  return network_write_chunk (request_id, fd, file_id,
     node_id, stripe_id,
                                chunk_num, offset, buf,
                                   count);
}

ssize_t process_delete_chunk (uint32_t request_id, int
   file_id, int node_id, int stripe_id, int chunk_num) {

  return network_delete_chunk (request_id, file_id,
     node_id, stripe_id, chunk_num);
}
```

Figure 6.2: Caching Lab Base Code

The caching lab requires students to write a caching data structure to store chunks deemed important into an in-memory data structure which includes the chunks and metadata information. Students can use other data structures to record statistics about which chunks are or are not frequently read. In the *process_read_chunk* function, if the chunk being read exists in the cache, the *network_read_chunk()* function call should be bypassed and the chunk data from the cache should be returned instead. In order for the Barista Core Module to recognize this, the chunk data should be written to the *buf* parameter and the size of the data in the buffer should be returned. In the *process_write_chunk* function, students need to update the cache data to make sure that invalid chunk data is not still stored in the cache. If the *process_delete_chunk* function is called, students should make sure to remove the chunk from the cache if it is stored in the cache. However, If a cache entry of a deleted chunk is not deleted, the Barista Code Module will prevent chunks that do not exist to be read. Many caching techniques exist, such as Least Recently Used and Most Recently Used, and it is left up to the students and/or instructor of the class to choose a specific technique.

6.2.2 Instructor Evaluation

Validation of this lab is difficult as it involves measuring timings of reads and writes that could be affected by more than the students' implementation of the lab. One option to verify that caching does exist is to write and read some chunks to DecaFS, then take all Espresso Nodes offline. Now, when a client attempts to read data from the system, there should be some chunks that are still readable. Another way to check that students are correctly invalidating and updating cache entries is to write a chunk to the system and read the chunk many times. After it is clear that the chunk has been added to the cache, write new data to the chunk. Subsequent reads of the chunk should return the updated data. If the old chunk data is returned, the cache is not updating cache entries properly.

6.3 Storage and Recovery Performance

The Storage and Recovery Lab has students explore how the system handles situations where Espresso Nodes go offline and are brought back online. If an Espresso Node goes offline for a period of time, then comes back online, the chunks that were updated while the node was offline need to be updated and synchronized to reflect the changes that occurred while the node was offline.

Learning Objective: To have students program a solution to update data on nodes that are out of date due to node outages

6.3.1 Student Implementation

When a node fails (goes offline) or returns (comes online), the Barista Core Module notifies the Monitored Strategy Module by calling the *node_failure_handler_func()* or *node_up_handler_func()* functions. These functions give students the opportunity to add functionality to DecaFS to handle system changes more elegantly than the default solution.

```
void node_failure_handler_func (uint32_t node_number) {
  printf ("Handling␣node␣failure...\n");
  // Add custom processing to be done when node
     node_number goes down.
}

void node_up_handler_func (uint32_t node_number) {
  printf ("Handling␣node␣coming␣online...\n");
  // Add custom processing to be done when node
     node_number goes up.
}
```

Figure 6.3: Recovery Lab Base Code

When the *node_failure_handler_func* is called, students should mark that the node is down and begin tracking modified chunks as invalid. When the node comes online and the *node_up_handler_func* is called, the invalidated chunks need to be updated to reflect the changes that occurred while the node was offline. In a mirrored and striped system, the data updates are simple as the data in the invalidated chunk can be overwritten with the data from its counterpart chunk (the replica chunk and primary chunk are each others counterpart). In a RAID 4 system, this is more difficult as students need to reconstruct the invalid chunk by computing the parity of all other chunks in the stripe. The solution to this lab also requires students to modify the I/O Manager Module to record which chunks are invalid. This requires small amounts of code to be added to the *process_read_stripe*, *process_write_stripe*, and *process_delete_file* functions.

6.3.2 Instructor Evaluation

In order to test this lab, the tester needs to force the updating of data that is stored on offline nodes. To do this, data should be written to the DecaFS instance, then *decafs_file_stat()* should be called to retrieve the Espresso Node id where the primary chunk is stored. This node will then need to be taken offline. Once offline, the chunk should be overwritten with new data. Bring the node online and read the chunk to ensure that the data returned from the last read is the same as the newly written data. The primary chunk is taken offline so that DecaFS is forced to write the new data to the replica chunk.

6.4 Adaptive Data Migration

The goal of this lab is to explore how to adjust the distribution strategy when an Espresso Node is overloaded with requests and cannot keep up with client demands. An uneven request distribution can drown a single Espresso Node and reduce the throughput of the system. To adapt to client demands, chunks can be read from idle Espresso Nodes to distribute the load across the system.

Learning Objective: To have students monitor system load and adapt the distribution of requests to improve system response times and better serve clients

6.4.1 Student Implementation

The first step of this lab is for students to develop a monitor that recognizes when an Espresso Node is being flooded with too many requests. This can be done by monitoring the read and write requests in the I/O Manager Module using the *process_read_stripe*, *process_write_stripe*, and *process_delete_file* functions. Once an imbalance has been recognized, students need to use the gathered metrics to decide

which replica chunks on idle nodes should be used. Once chunks have been identified, the primary and replica nodes of the chunk should be swapped. This can be achieved by calling the *set_node_id* and *set_replica_node_id* functions from the I/O Manager Module with the identification information of the chunk.

```
int set_node_id (uint32_t file_id, uint32_t stripe_id,
   uint32_t chunk_num, uint32_t node_id);
int set_replica_node_id (uint32_t file_id, uint32_t
   stripe_id, uint32_t chunk_num, uint32_t node_id);
```

Figure 6.4: Data Migration Lab Function Declarations

6.4.2 Instructor Evaluation

Validation of this lab requires the creation of a situation where data migration is necessary. To do this, there needs to be a DecaFS Client that constantly reads data from the system. In the RAID 10 base implementation, this will cause the Barista Core Module to read from the Espresso Nodes storing the primary chunks. After a short period of time, the students implementation should see a need to adapt the distribution of the primary and replica chunks. Once these primary and replica chunk labels have been distributed, a call to *decafs_file_stat* of all files in the DecaFS instance will return data showing that all Espresso Nodes are storing both primary and replica chunks.

6.5 RAID 4

A more advanced challenge for students to implement would be to change from the RAID 10 striping and mirroring solution implemented in the Distribution and Replication lab to RAID 4, a scheme that increases the capacity of the file system without

sacrificing fault-tolerance. RAID 4 is a scheme with chunk striping and a dedicated parity disk.

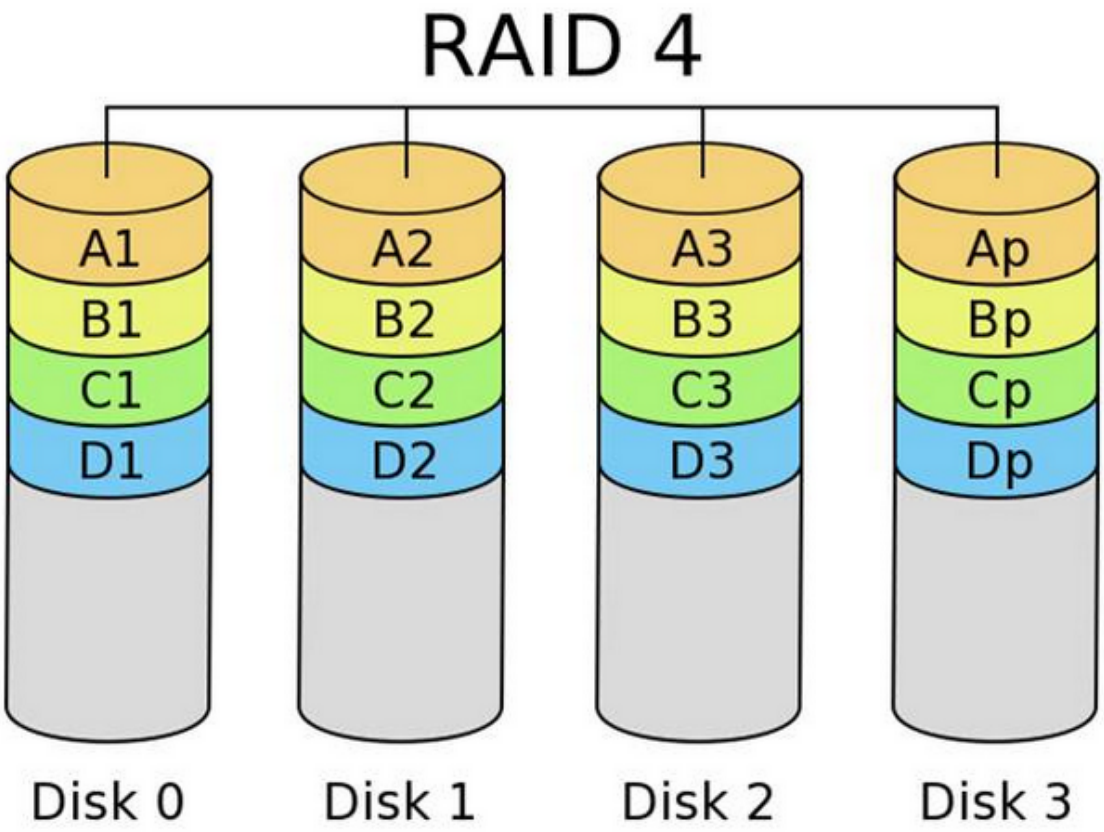

Figure 6.5: RAID 4 Chunk Distribution [13]

Instead of having a dedicated replica chunk per primary chunk, a special type of replica chunk called a parity chunk is used to provide fault tolerance.The parity chunk is computed by taking the XOR of all chunks in a stripe. By computing the parity chunk in this way, any chunk of the stripe can be re-created by taking the XOR sum of all other chunks in the stripe.

1. $Ap = A1XORA2XORA3$
2. $A1 = A2XORA3XORAp$

Learning Objective: To have students implement a more complex fault-tolerant data storage scheme that prioritizes capacity over performance

6.5.1 Student Implementation

This lab requires students to modify three modules in DecaFS: Distribution, Replication, and I/O Manager. In the Distribution Module, students need to split a stripe into three chunks and distribute them to the corresponding disks or nodes in the system. In the Replication Module, a parity chunk needs to be computed and stored on the parity disk. Read requests are simple when no nodes have gone down in the system, but if one of the non-parity nodes goes down, students have to read all remaining chunks and compute their XOR sum as described above. This operation would be slow as you would have to read data from all three nodes and compute the result of the returned data before you can return the requested data to the client. This is one of the most compelling labs because it uses Raid 4 which NetApp used in their WAFL file system.

6.5.2 Instructor Evaluation

Evaluation of this lab can be performed relatively easily. Write a file to the DecaFS instance and call *decafs_file_stat* to see how the file is distributed across the Espresso Nodes. A stripe of the file should have each of its three chunks written to three different Espresso Nodes and the replica chunk of the stripe should reside on the other Espresso Node. Additionally, the Espresso Node where the parity chunk is stored should have one replica chunk per stripe stored on the file system. The validation also needs to ensure that students have correctly computed the parity chunks of the stripes. To verify this, each of the Espresso Nodes in the system should be shut down, one at a time, and the entire stripe should be read. If the stripe is correctly recreated

with any single Espresso Node offline, the parity chunk has been computed correctly.

6.6 MapReduce

This lab is the only lab described that does not rely on DecaFS, instead it uses the Raspberry Pi cluster and the OpenMPI library that is installed on the Raspberry Pi computers. MapReduce is a programming model designed for use with large sets of data. Simple implementations using MapReduce can digest large amounts of data to transform or simplify the data set. The MapReduce exercise suggested for this lab is to have students create a histogram of the sum of two vectors.

Learning Objective: To have students implement a simple distributed program to familiarize them with two common distributed computing tools, MapReduce and MPI

6.6.1 Student Implementation

Using the simple MPI functions of Init, Comm_size, Comm_rank, Barrier, Finalize, Send, and Recv, students can build a MapReduce solution to compute a histogram of the sum of two vectors. A student solution may look similar to the following:

1. Setup the MPI calls required to synchronize nodes
2. Read the two vectors into memory on the master node
3. Distribute the vectors across nodes using the index of the value as its key
4. For each set of pairs with identical keys, compute the sum of the values
5. Add the sum to a local histogram
6. Send all partial histograms to the main node

7. Sum the partial histograms together to create the final histogram

6.6.2 Instructor Evaluation

Evaluation of this lab is simple, the MapReduce version of the program should produce the same histogram as a serial implementation of the algorithm. If additional work is desired, students can also write a serial version and compare the timing results of the two algorithms together.

6.7 Proposed Lab Layout

The labs in this Chapter represent a variety of distributed learning objectives that can be included in a distributed computing class. We recommend that the DecaFS labs be administered in the order they are presented:

Lab	Estimated Effort (1 easy - 5 hard)	Hours	Duration (weeks)
1. Distribution and Replication	1	6	1
2. Caching	2	4	1
3. Storage and Recovery Performance	4	10	2
4. Adaptive Data Migration	3	8	1.5
5. RAID 4	5	15-20	2.5
6. MapReduce	3	8	1.5

Table 6.1: Estimated Timeline

Overall, the six labs should take an estimated ten week duration to complete using the estimations displayed in Table 6.1. This ten week class calendar would help students become familiar with core distributed computing concepts (the labs). All labs are extensions of the base functionality we provide, so any lab can be omitted

from a class schedule. The information students learn along the way is useful for future labs, but implementation of the previous lab is not necessary. The MapReduce lab stands independent of DecaFS and can be incorporated into the class if the instructor feels the learning objective of the lab is important for students to grasp. If all six labs are assigned within a quarter, it is unlikely that additional labs would fit into the schedule; however, the labs could be supplemented with a quarter long project on a student selected topic in the area of distributed computing.

CHAPTER 7

Testing and Validation

This chapter describes the testing and validation performed for this project. We approached testing with a bottom up method by beginning with unit tests, progressing to integrated tests, and finishing with system tests. We did not conduct user acceptance tests as that is considered part of the activity that will occur when the platform is used to support a distributed programming curriculum in Cal Poly classes. For the Raspberry Pi Clusters, we conducted integrated testing for the individual Raspberry Pi computers and systems testing for each of the Raspberry Pi Clusters. For DecaFS, we unit tested individual modules, then conducted integrated testing for each layer, and finished with a set of system tests to ensure the system worked as specified. To test modules and layers of the system, we used Google Test and Google Mock to remove dependencies for more effective testing. No validation of the labs are provided as this step is pending their deployment in classes at Cal Poly.

7.1 Terms and Definitions

The following terms are used throughout this Chapter, for ease of reading, the terms are defined here.

- **Fake** - "objects actually have working implementations, but usually take some shortcut which makes them not suitable for production" [19]
- **Stubs** - "provide canned answers to calls made during the test, usually not

responding at all to anything outside what is programmed for the test. Stubs may also record information about calls" [19]

- **Mock** - "objects are pre-programmed with expectations which form a specification of the calls they are expected to receive" [19].

7.2 Google Test and Google Mock Tools

Google Test is a C++ testing framework that provides users the capability to quickly write tests using built-in assertions and test fixtures [15]. Google Mock is a mocking framework that can be used with Google Test or any C++ testing framework. Google Mock allows users to easily create mock objects where values can be specified to easily produce desired states and circumstances in the system [14].

7.3 Test Plan

7.3.1 Raspberry Pi and Raspberry Pi Cluster

The individual Raspberry Pi computers are not unit tested as the boards come tested from the manufacturer. Integrated tests are performed on each Raspberry Pi computer configured with their SD card, operating system image, wireless dongle, and power supply are tested to verify that they work properly with all software and hardware components. The Raspberry Pi Clusters are also tested as a system to ensure that all nodes can connect and communicate with each other over the network.

7.3.2 DecaFS

Modules that include base functionality of DecaFS are unit tested to ensure that the implementation follows the design specification of the module. Modules with

dependencies on other modules use mocks to allow them to be unit tested independently. The unit tests of these modules validate the implementation of the module by mocking its dependencies and setting expectations for which functions should be called, how they should be called, and what they will return. The Espresso Layer is the easiest to unit test because the modules in this layer have few or no dependencies that need to be mocked. The Network Layer is also unit tested, but it uses a Fake implementation of the Espresso Layer that stores and reads chunk data from an in-memory data structure.

After modules have been unit tested, we proceed to test groups of modules working together. We grouped the units together and tested them as a whole. These integration tests are performed at the layer level as we found that to be the most logical functional grouping of the modules. The goal of these tests is to validate that the modules operate correctly as a whole. These tests know nothing about the inner workings of the layer; they just test the functionality of the exposed API using inputs and outputs.

The last type of tests we conducted were system tests. These tests validate the system as a whole and do not Mock, Fake, or Stub any part of the testing. For DecaFS, system tests ensure that the file system correctly implements the API that we provide to DecaFS Clients. To execute system tests, DecaFS is setup on a five node cluster of Raspberry Pi computers and a DecaFS Client is run on one of the five nodes. The DecaFS Client is used to test file system functionality such as reading, writing, opening, and deleting of files. These tests also verify that DecaFS provides functionality like single-node fault tolerance as designed.

7.3.3 Labs

At this time, testing has not been conducted on the labs as testing is pending the deployment of the entire platform in a class at Cal Poly. At that time, user acceptance testing will be performed to evaluate their effectiveness in the classroom and the usability of the DecaFS API.

7.3.4 Summary

	Unit Tests	Integration Tests	System Tests	Acceptance Tests
Cluster		x	x	
DecaFS	x	x	x	
Labs				Future

Table 7.1: Testing Matrix

7.4 Unit Tests

7.4.1 DecaFS

7.4.1.1 Volatile Metadata Module

This is a standalone module and can be tested independently. Scenarios:

- chunk size can be set
- chunk size cannot be set to an invalid size
- chunk size cannot be reset
- stripe size can be set
- stripe size cannot be set to an invalid size

- stripe size cannot be reset
- number of active nodes can be queried
- node ids can be queried for their status (online / offline)
- offline nodes are not present in the active node list
- offline nodes cannot be set to offline
- file cursors can be created
- file cursors can be set
- file cursors can be deleted
- file cursors that do not exist return an error when queried
- volatile file metadata can be queried

7.4.1.2 Persistent Metadata Module

This is a standalone module and can be tested independently. Scenarios:

- files can be added
- number of files can be queried
- file names can be queried
- persistent file metadata can be queried
- file access time can be set
- file size can be set
- files can be deleted

7.4.1.3 Locking Strategy Module

This is a standalone module and can be tested independently. Scenarios:

- exclusive lock can be acquired
- exclusive lock can be released
- exclusive lock can be queried
- multiple clients cannot acquire an exclusive lock on the same file at the same time
- multiple processes of a client cannot acquire an exclusive lock of the same file at the same time
- shared lock can be acquired
- shared lock can be released
- shared lock can be queried
- multiple clients cannot acquire a shared lock on the same file at the same time
- multiple processes of a client can acquire a shared lock of the same file at the same time

7.4.1.4 Distribution Strategy Module

This is a standalone module and can be tested independently. Scenarios:

- chunks are striped across the two primary nodes
- primary chunks are not stored on the replica nodes

7.4.1.5 Replication Strategy Module

This is a standalone module and can be tested independently. Scenarios:

- replica chunks are striped across the two replica nodes
- replica chunks are not stored on the primary nodes

7.4.1.6 Storage Module

This is a standalone module and can be tested independently. Scenarios:

- chunks can be allocated
- chunks can be written
- chunks can be overwritten
- chunks can be appended to
- chunks can be read
- chunks can be deleted
- non-existent chunks cannot be read
- non-existent chunks cannot be deleted
- a negative size cannot be read
- a negative size cannot be written
- disk space of deleted chunks are freed
- chunk data is persistent

7.5 Integration Tests

7.5.1 Raspberry Pi Computer

After a Raspberry Pi computer has been setup by installing the Raspbian operating system on the SD card and plugging in the wi-fi dongle, the following tests are run to verify that that the computer is functioning properly. Scenarios:

- the network can be accessed
- files can be created
- files can be modified
- files can be read
- files can be deleted

7.5.2 DecaFS

7.5.2.1 Barista Layer

The Barista Layer is the highest layer in DecaFS and independent of both the Network and Espresso Layers. To remove the Barista's dependencies on the other layers for testing purposes, we implemented a fake that mimics the functionality of the lower layers in the system. The API the Barista interacts with is concerned with making sure that chunks can be written and retrieved from a specific Espresso Node in the file system. To do this, we used a `map` data structure to match chunk identifiers to the chunk's data. This can be done very easily as the Network API already contains all of the information to identify a chunk. See Figure 6.2 for the information that the Network Layer receives with a request from the Barista. To store this in a `map`, we

needed to convert a series of identifiers (file id, node id, stripe id, and chunk number) into a string that could be used as a key into our map. We concatenated all identifier fields separated by a period to distinguish the different fields. With this fake we tested the Barista Node without setting up the entire system.

Tests at the Barista Layer are concerned with the interactions of the modules within the Barista Layer. If we have implemented a round robin method, can we write multiple chunks and verify that they were written to the correct nodes using our custom *decafs_file_stat*? These tests are less concerned that the implementation works as a file system and more concerned with validating that the system does in fact mirror and replicate or follow a specific RAID strategy.

Scenarios:

- files can be created
- files can be opened
- files can be written to
- files can be read from
- files can be deleted
- file metadata can be retrieved
- multiple processes of a client can read from the same file
- multiple clients cannot open the same file (for reading or writing)
- filedata is persistent

7.5.2.2 Network Layer

The Network Layer is responsible for reliably sending data between the Barista Node and Espresso Nodes in the system. To remove the dependencies of the Network Layer, we needed to create a fake of the Espresso Layer. Scenarios:

- all packet types can be serialized and deserialized
- all packet types can be transferred across the network without corruption
- data can be written across the network
- data can be read across the network
- client requests are translated and call the correct Barista Core Module functions

7.5.2.3 Espresso Layer

The unit tests in Section 7.4.1.6 are sufficient for the integration tests of this layer because the layer is composed of only two modules.

7.6 System Tests

7.6.1 Raspberry Pi Cluster

Each Raspberry Cluster was tested to ensure that each Raspberry Pi in the cluster could communicate with all other Raspberry Pi computers in the cluster over the network.

- each Raspberry Pi can ping all other Raspberry Pi computers in the cluster

7.6.2 DecaFS

The system tests performed on DecaFS are a superset of the tests performed for the integration testing of the Barista Layer. These tests also include tests that verify that files can still be read from when one of the nodes storing the data to be read is offline. These tests are done on a cluster with a production instance of DecaFS deployed.

Scenarios:

- files can be created
- files can be opened
- files can be written to
- files can be read from
- files can be deleted
- file metadata can be retrieved
- multiple processes of a client can read from the same file
- multiple clients cannot open the same file (for reading or writing)
- filedata is persistent

CHAPTER 8

Conclusions

As stated in the contributions section of this thesis Section 1.2, the overarching goal of the work covered in this project was to create a complete educational framework for the teaching of distributed computing at Cal Poly. With the design, implementation, and testing of the Raspberry Pi Cluster, DecaFS, and Labs laid out in the main chapters of this work, the framework for such a class has been completed. A class in distributing computing could compile a set of the labs laid out in CHAPTER 6 to introduce students to distributed concepts and familiarize them with the Raspberry Pi Cluster and DecaFS. As the class progresses, more advanced labs could be assigned as larger projects or students could be directed to create their own distributed computing programs on the Raspberry Pi Cluster.

8.1 Raspberry Pi Cluster

In CHAPTER 4, the five node Raspberry Pi clusters designed for this project are described. These clusters were designed to support distributed computing classes at Cal Poly. The requirements were as follows:

1. 8 clusters with minimum of 5 nodes per cluster
2. Physically small enough to be portable classroom to classroom
3. Modular in order to assemble / disassemble as desired

4. Affordable within the budget of CP-Connect grant (approximately $4000)

5. Scalable in quantity of nodes and additional clusters

6. Use of free open source software (operating system and tools)

By creating the clusters using Raspberry Pi computers, we were able to create small portable clusters within the limited budget allocated. Each of the eight clusters is composed of five Raspberry Pi computers that connect to a wireless router. As the computers are about the size of a credit card, an entire cluster can fit into a shoebox. These small clusters are easily modifiable and scalable as nodes can be added by plugging in another Raspberry Pi. The clusters use the free Raspbian operating system and have a small group of extra tools installed to support student development and testing.

8.2 DecaFS

CHAPTER 5 covers the design and implementation of a modular distributed file system that is described in detail in my colleague's work [10]. The DecaFS distributed file system was designed to support the labs described in CHAPTER 6 by implementing the core of the file system in a modular manner, separating the file system and network code from the distributed portion of the system. The modular design isolates code through the use of APIs and layers. This allows students to implement the labs without dealing with the file system and network code layer. Each lab described in CHAPTER 6 can be completed by modifying at most three modules of the system. These modules only contain code used to distribute and recover data thereby providing hands-on experience with distributed programming to the students.

8.3 Labs

The Labs defined in this thesis were created to represent a set of topics that cover a wide variety of problems encountered in distributed systems. These labs cover the topics of distribution, replication, fault tolerance, recovery, rebalancing, and efficiency. While these labs represent a wide range of problems in distributed computing, this project does not validate the use of these labs in a classroom setting as no classes have yet been offered that use the Raspberry Pi cluster, DecaFS, and the labs laid out in this thesis.

8.4 Summary

With this project, a framework for distributed computing curriculum at Cal Poly has been created and delivered. As classes are offered that use this framework of the Raspberry Pi Clusters, DecaFS, and Labs, the framework and curriculum will need to continue to evolve to keep pace with the ever advancing information technology world so that it may continue serving the needs of the faculty and students of Cal Poly.

CHAPTER 9

Future Work

Overall, we accomplished the goals we set out to meet at the beginning of this project; however, there is still room for improvement of the system. Due to time constraints, the work described in this Chapter was not incorporated into the work of this thesis.

9.1 Classroom Usability

The next step for the Raspberry Pi Clusters, DecaFS, and the Labs is to use them in the distributed computing classes they were designed for. Once some of the labs described in CHAPTER 6 have been completed by students, it would be useful to start a dialogue with the students to gather feedback on what could be done to improve the usability of the system for the class.

9.2 Validation Tools

One area for future work on the DecaFS filesystem is in the validation area. Students would benefit from the creation of visual tools that connect to the system and display current metrics. Distribution statistics such as amount of data stored on each node with expandable lists containing the details of the chunks stored on a specific node: file_id, stripe_id, and chunk_id is one example. Other data points like network usage and node status could also be added to a debugging tool such as this. Having a tool like this available to students would help them visualize how their code is actually

distributing data and may even be useful in the instructor's evaluation of student work.

9.3 Testing

CHAPTER 7 describes the testing methodology and code sections tested in the current code base. While we tested the modules of DecaFS and made sure that the base functionality we provide works correctly, we do not provide a way for students or faculty to test the labs the system was designed to support. One area for future work is to supply a set of system tests that can be used to verify the correct implementation of each lab. This would require a modified testing suite for each lab as well as one for the base functionality we provide. A second item to enhance the testing area would be to measure the performance of the system. This would be useful for students and faculty as they could run performance benchmarks to assess the speed at which standard filesystem operations complete in their modified systems. This test suite would include tests of random and sequential read and write operations of different sizes.

BIBLIOGRAPHY

[1] ARM1176 Processor. http://www.arm.com/products/processors/classic/arm11/arm1176.php. Accessed: 2015-02-24.

[2] Raspberry Pi Model B. http://www.raspberrypi.org/products/model-b/. Accessed: 2015-02-26.

[3] Raspberry Pi Models and Revisions. http://www.raspberrypi.org/documentation/hardware/raspberrypi/models/README.md. Accessed: 2015-02-24.

[4] What is a Raspberry Pi? http://www.raspberrypi.org/help/what-is-a-raspberry-pi/. Accessed: 2015-02-24.

[5] Apache Hadoop. HDFS Architecture Guide. http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.

[6] J. Cappos, I. Beschastnikh, A. Krishnamurthy, and T. Anderson. Seattle: A platform for educational cloud computing. *SIGCSE Bull.*, 41(1):111–115, Mar. 2009.

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

[8] K. Hoganson. Computer Science Curricula in a Global Competitive Environment. *J. Comput. Sci. Coll.*, 20(1):168–177, Oct. 2004.

[9] J. Kiepert. Creating a Raspberry Pi-Based Beowulf Cluster. Technical report, Boise State University, 05 2013.

[10] H. Meth. DecaFS: A Modular Distributed File System to Facilitate Distributed Systems Education. Master's thesis, California Polytechnic State University, San Luis Obispo, CA United States, 2014.

[11] Oracle, Intel Corporation. Lustre File System Manual. https://build.hpdd.intel.com/job/lustre-manual/lastSuccessfulBuild/artifact/lustre_manual.xhtml.

[12] Raspberry Pi forum member bwann. Hadoop + HDFS + MR on Pi cluster - works! http://www.raspberrypi.org/forums/viewtopic.php?t=37190.

[13] Recovery Italia. Procedura di recupero dati da RAID 4. http://www.raidrecovery.it/raid-knowledge/livelli-raid/recupero-dati-raid4/.

[14] Google, Inc. Google Mock. https://code.google.com/p/googlemock/.

[15] Google, Inc. Google Test. https://code.google.com/p/googletest/.

[16] 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.

[17] Guay, Patrice. An Overview of RAID technology. http://blog.iweb.com/en/2010/05/an-overview-of-raid-technology/4283.html.

[18] Intel Corporation. Why Use Lustre. https://wiki.hpdd.intel.com/display/PUB/Why+Use+Lustre.

[19] Martin Fowler. Mocks Aren't Stubs. http://martinfowler.com/articles/mocksArentStubs.html.

[20] A. Rotem-Gal-Oz. Fallacies of Distributed Computing Explained. Technical report, 2006.

[21] A. S. Tanenbaum and M. v. Steen. *Distributed Systems: Principles and Paradigms (2Nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[22] P. Weygant. *Clusters for High Availability: A Primer of HP Solutions*. Pearson Education, 2001.

APPENDIX A

APIs

A.1 Network Core API

```
/* sends a read chunk request to an espresso node
 * returns -1 on error
 */
int network_read_chunk(int32_t id, int fd, int file_id,
   int node_id, int stripe_id,
 int chunk_num, int offset, int count);

/* sends a write chunk request to client
 * returns -1 on error
 */
int network_write_chunk(int32_t id, int fd, int file_id,
   int node_id, int stripe_id,
 int chunk_num, int offset, void* buf, int count);

/* sends a delete chunk request to client
 * returns -1 on error
 */
```

```
int network_delete_chunk(int32_t id, int file_id, int
   node_id, int stripe_id, int chunk_num);
```

A.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
16 struct 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);
```

A.3 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);
```