

Programming Assignment 3:

A Simple Distributed File System

BY: Julia Garbuz

garbu007@umn.edu

ID: XXXXXXXXXX

Table of Contents:

0	Table of Contents	2
1	Introduction	3
1.1	Purpose	
2	Design	3
2.1	System Design	
2.2	DFS Configuration	
2.2.1	Servers	
2.2.2	Quorum Selection	
2.2.2	Background Update Frequency	
2.3	Component Design	
2.2.1	DFS Coordinator Node	
2.2.2	DFS (Regular) Node	
2.2.2	Client	
3	Usage	7
3.1	Starting the DFS	
3.1.1	Configuring the DFS	
3.1.2	Starting Coordinator Node	
3.1.3	Starting (Regular) Nodes	
3.1.4	Starting Client	
3.2	Interacting with the DFS	
3.2.1	DFS CLI (via Client)	
3.3	Cleaning and resolving issues	
3.3.1	Clear	
3.3.2	Killing DFSNodeInstanc(es)	
3.3.3	Rebuilding Thrift Files	
4	Testing and Results	15
4.1	Negative Tests	
4.2	DFS Performance Tests	

1 Introduction

1.1 Purpose

The goal of this project was to implement a simple distributed file system (or DFS) using Apache Thrift. The file system allows users to read or write files to the same file system via multiple concurrent clients. The files within the system are replicated over multiple servers who are coordinated by a “coordinator” node. The coordinator node implements quorum-based protocol to manage consistency.

2 DESIGN

2.1 System Design

The basic construction of the system can be seen in **Diagram 2.1**. The DFS must consist of a minimum of seven servers, one of which also functions as the coordinator node. Any of the servers can receive requests from any client to the system, but they must all forward their requests through the coordinator node to assemble a quorum for voting on file versions. The details of quorum construction can be found in [Section 2.2.2](#). Once a result has been agreed on, that result is sent back to the server who forwarded the request and then back to the client.

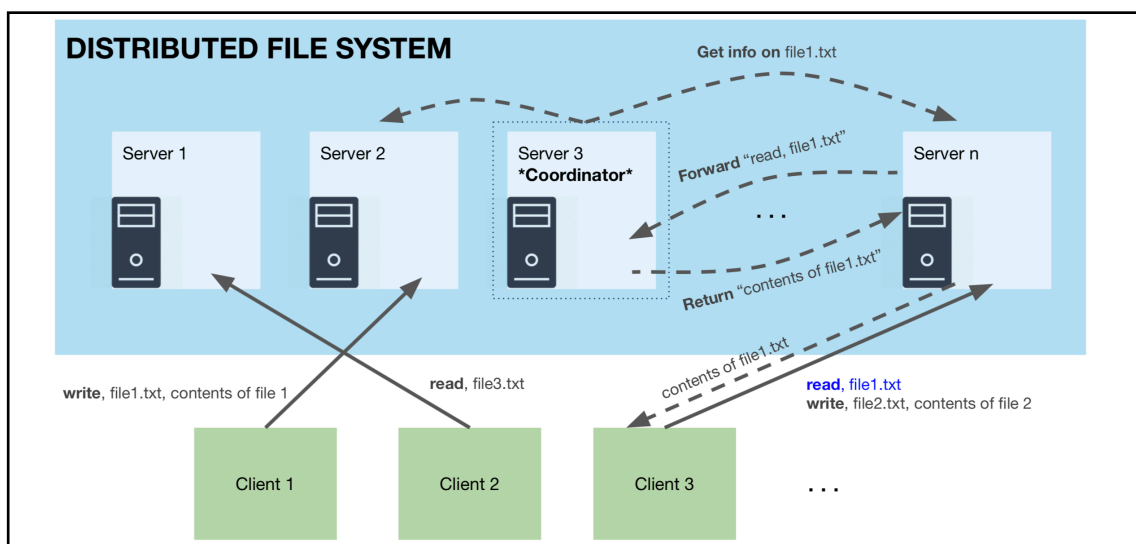


Diagram 2.1: Basic System Diagram

2.2 DFS Configuration

Despite not being thoroughly fleshed out, the current working version of the DFS allows for quite a bit of configuration. Details on how to configure this settings can be found in [Section 3](#). For now we shall discuss the design behind the configurability.

2.2.1 Servers

The DFS will work with any number of nodes above or equal to seven as long as one of the nodes has been constructed as a coordinator node. These servers can be independent machines or merely running on different ports on the same machine.

Both the coordinator node and the other DFS nodes' functionalities are implemented by **DFSNodeHandler** since the coordinator is merely a DFS node with additional responsibilities. These additional responsibilities (i.e. quorum assembly) are handled by a **DFSCoordinator** object. The universal responsibilities of local file management (reading, writing, locking, version tracking, etc), on the other hand, are handled by the **FileManager**. Depending on the arguments passed to the **DFSNodeInstance**, the **DFSNodeHandler** will be configured either as a coordinator or regular node.

For the sake of ease of usability, the system requires that the coordinator node be started first and it writes its information to a globally accessible properties file. This is done, so that when other DFS nodes or clients are started, they can automatically read the coordinator's information to connect to the system. In this manner, the coordinator node also immediately collects the IP and port information on all of the other nodes trying to join the DFS which it will later use to assemble quorums.

2.2.2 Quorum Selection

For situationally improved performance as well as experimental testing, several options have been provided to configure how the quorums are assembled. There are five settings for quorum selection:

- USER_CONFIG**: Allows user to manually set N, Nw, and Nr
- RANDOM**: Randomly selects valid Nw and Nr, given N
- READ_HEAVY**: Minimizes Nr and maximizes Nw, given N
- WRITE_HEAVY**: Minimizes Nw and maximizes Nr, given N
- CONSISTENT**: Sets Nw and Nr to N, so all nodes always up to date

The effects of the last four on performance are thoroughly analyzed in [Section 4.2](#). These options, like many of the other configurations have defaults, but can

also be set in the **dfs.properties** file or passed as command line arguments when starting the coordinator node. Details on how to do this are provided in [Section 3.1.1](#).

2.2.3 Background Update Frequency

The last largely configurable component of the DFS is the frequency of the nodes' background updates. This was provided because some use cases might require or prefer more quickly “eventually consistent” data. It is important to note that as of now, a singular frequency is set for all nodes in the DFS. In the future the option could be provided to set it on a node-by-node basis. Again, details on configuring the frequency can be found in [Section 3.1.1](#).

2.3 Component Design

The following sections will describe the design and functionality provided by each of the main components in more detail.

2.3.1 DFS Coordinator Node

The DFS coordinator node, as previously mentioned, performs two roles: that of the coordinator and that of a “regular” DFS node. In this section we will only discuss the first. The latter will be described in [Section 2.3.2](#) below.

The handler of the coordinator node's functionality is the same as that of a regular node (**DFSNodeHandler**) but some functions are just handled differently. For example, when a “read()” request is received. If the instance of **DFSNodeHandler** receiving the request is not a coordinator node then it will forward it to the coordinator. Otherwise, the coordinator begins assembling the read quorum. In the following sections the implementation of read and write as handled by the coordinator are described.

2.3.1.1 Reads

When the coordinator receives a read request, it first acquires a lock on that file name (via **DFSCoordinator**) and in this manner queues the requests in the order they arrive on a file-by-file basis so that everything is evaluated as intended.

Next, the **DFSCoordinator** handles (1) assembling a random read quorum (based on *Nr* determined by configuration) (2) querying every node in the quorum for version information on the requested file (3) selects one of the nodes with the most recent version of the requested file.

At this point, the coordinator node opens a connection to this node and makes a **performRead** request, releases the lock on the file, and returns the result of the sub-request to its caller.

2.3.1.2 Writes

Write is implemented very similarly to read except for that the **DFSCoordinator** only handles assembling a write quorum, and then the coordinator itself connects to the nodes in the write quorum and make **performRead** requests which it then returns the success or failure of to its caller.

2.3.2 DFS (Regular) Node

The DFS node key points of functionality are (1) forwarding requests to the coordinator node and returning the results to the client, (2) managing its local files, and (3) periodically performing updates to achieve eventual consistency.

The local file management is performed by the **FileManager**. The **FileManager** manages the locks on the files, versions, and the physical reading and writing.

The updates are called on a pre-configured frequency in the background by a thread of **DFSNodeInstance**. Updates merely make a read request for every one of the files the **FileManager** contains, and if the version received is newer than its own, then it writes the update file information to the local system.

The local file system is physically written to a data directory and organized by folders per node (to avoid overwriting on the CSE Machines).

2.3.3 Client

The client's sole responsibility is handling the user input on the CLI, forwarding these requests to the node it is connected to, and then displaying the results of the request to the user.

One important thing to note is that the node the client connects to is randomly "assigned" to it via a **getRandomNode** request that the client makes of the coordinator upon starting. Alternatively, as will be seen in [Section 3.1.4](#), if the user is aware of one of the DFS nodes' IP and port, it can provide this as command line arguments and the client will connect to that machine immediately.

Once configured and connected, the client provides a CLI into the DFS. The four main functionalities provided by the CLI are:

WRITE: Write provided contents to a file in DFS

READ: Read contents and version of file in DFS

LIST FILES: List most recent versions of all files on DFS**

*** It is important to note that this can display files or versions not on the node currently connected to, but will NOT update the nodes information.*

SUBMIT: Submit a file of any number of these requests (batch processing)

More detail on their options and usage is provided in [Section 3.2](#).

3 Usage

3.1 Starting the DFS

There are three main components to starting the DFS: the coordinator, the nodes, and the client for interacting with the DFS.

It is **highly recommended** you start the system in one of the following two ways (order of starting components is of importance, and please see the **NOTE** in [Section 3.1.3](#) in regard to the timing of starting the scripts):



Figure 3.1: Two ways of starting DFS

Although the `./start-node`, `./start-nodes-on-ports`, and `./start-client` can be run without a coordinator, their functionality depends on the coordinator running and the coordinator start script includes some prep and clean up that the other start scripts do not contain (which could raise errors in the future).

The following sub-sections will go into detail on how to configure and start each component.

3.1.1 Configuring the DFS

There are two ways to configure the DFS:

- (1) via the **properties file** (`src/main/resources/dfs.properties`)
- (2) via **command line arguments** to the coordinator's start script.

***NOTE:** All of the parameters are configurable both ways EXCEPT for `update_frequency` which can only be changed in the properties file.

The DFS configurations are assembled from the following hierarchy (each inheriting from and overloading the one below it):

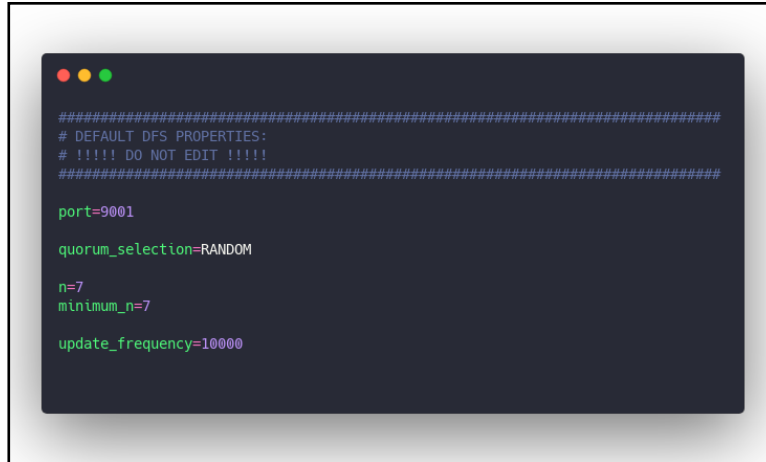
- ↳ Command line arguments to `./start-coordinator`
 - ↳ `src/main/resources/dfs.properties`
 - ↳ `src/main/resources/default.properties`

This means that if no properties are set by the user, all of the properties from `default.properties` are used. Additionally, if the user configures something invalid, the defaults will be used (i.e. an invalid N, Nw, Nr combination).

3.1.1.1 Default DFS Configurations

The contents of `default.properties` is provided in [Figure 3.1.1.1](#). These values are not for editing, but are merely provided in the document for user reference.

The `port` field in `default.properties` is used as the default port for the coordinator node only.



```
#####
# DEFAULT DFS PROPERTIES:
# !!!!! DO NOT EDIT !!!!!
#####

port=9001

quorum_selection=RANDOM

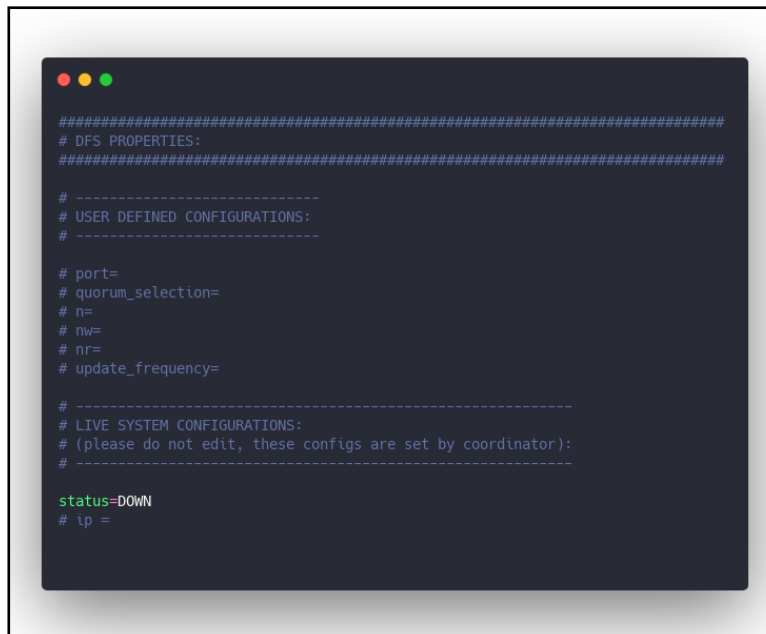
n=7
minimum_n=7

update_frequency=10000
```

Figure 3.1.1.1: Contents of default.properties

3.1.1.2 Configuring the DFS via Properties File

The initial contents of `dfs.properties` is provided in [Figure 3.1.1.2 a](#). All of the values you can configure are listed here as comments under the heading “USER DEFINED CONFIGURATIONS”. The values under the heading “LIVE SYSTEM CONFIGURATIONS” are written to by the coordinator when it is ready to be connected to, and read by all other nodes and clients. Do not edit them.



```
#####
# DFS PROPERTIES:
#####

# -----
# USER DEFINED CONFIGURATIONS:
# -----

# port=
# quorum_selection=
# n=
# nw=
# nr=
# update_frequency=

# -----
# LIVE SYSTEM CONFIGURATIONS:
# (please do not edit, these configs are set by coordinator):
# -----

status=DOWN
# ip =
```

Figure 3.1.1.2 a: Initial state of dfs.properties

Figure 3.1.1.2 b below provides information regarding the options for configuring each of the fields:

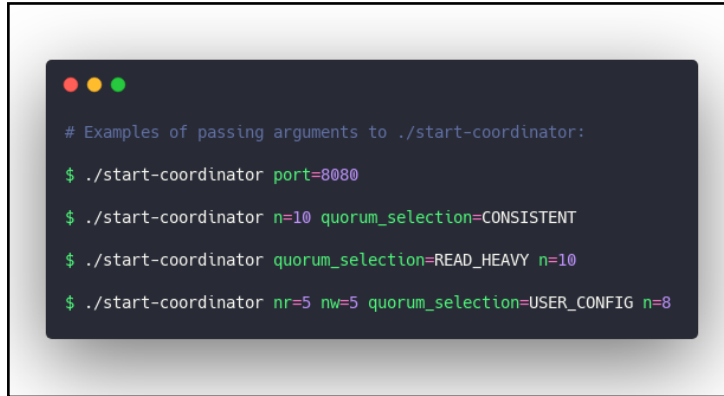
Configurable fields:	
port	<u>Port of coordinator node</u> Any valid and available port
quorum_selection	<u>Quorum selection setting</u> USER_CONFIG, RANDOM, READ_HEAVY, WRITE_HEAVY, or CONSISTENT See Section 2.2 for what each setting means
n	<u>Number of replicas/nodes</u> Any integer value ≥ 7 . Will reset to default of 7 if not. <ul style="list-style-type: none"> - Count includes coordinator node - Coordinator will not be “ready” till n nodes are in DFS
nw	<u>Nw</u> Any integer value such that: (1) $nr + nw > n$ (2) $nw > n/2$ If either condition not satisfied, will revert to default quorum_collection of RANDOM and randomly generate nw and nr .
nw	<u>Nr</u> See nw for conditions and notes.
update_frequency	<u>Frequency of background update</u> Any integer. In milliseconds. WARNINGS: (1) Does not have lower bound check. Recommended to keep above default of 10000. (2) Can ONLY configure in properties file (cannot configure via command line arguments)

Figure 3.1.1.2 b: Information configurable fields

When the system is shut down, all user defined configurations are left untouched except for potentially the port. If a new port is defined via the command line, it will over-write whatever what was listed in `dfs.properties` since the nodes and clients must read in the port information via the properties file. c

3.1.1.3 Configuring the DFS via Command-Line Arguments

All of the same fields (except **update_frequency**) can be set via command line arguments to **./start-client**.



```
# Examples of passing arguments to ./start-coordinator:
$ ./start-coordinator port=8080
$ ./start-coordinator n=10 quorum_selection=CONSISTENT
$ ./start-coordinator quorum_selection=READ_HEAVY n=10
$ ./start-coordinator nr=5 nw=5 quorum_selection=USER_CONFIG n=8
```

Figure 3.1.1.1: Examples of passing command-line arguments to coordinator

The arguments can be listed in any order as long as they are formatted as they are in the property files (“field=value”) with no spaces between the field, equal-sign, and value. [Figure 3.1.1.3](#) provides some examples.

3.1.2 Starting Coordinator Node

```
$ ./start-coordinator
```

The start script for the coordinator does several things including building Thrift files, compiling and starting the **DFSNodeInstance**, and then handling clean up upon exiting.

For which arguments you can provide to the script, see [Section 3.1.1.3](#).

It is important that you “quit” the coordinator node with control-C. This escape is caught so that the “live configurations” in the property file can be properly re-set.

If the coordinator node happens to crash or if you “quit” it differently, and now it refuses to restart, see [Section 3.3](#). You should simply need to run the “./clean” script (and “./kill-dfs-nodes” if instances are still running).

3.1.3 Starting (Regular) Nodes

NOTE: It is important that if you have started a coordinator, that you **wait until it prints that it is “starting”** (i.e. “[NodeInstance] Starting on 'cse1-kh4240-02.cselabs.umn.edu:9001'”) before starting any of these scripts. Otherwise their pre-compilation work will clash (compiling Thrift, etc).

```
$ ./start-node [ port ]
```

```
$ ./start-node [ port ] [ coord_ip:coord_port ]
```

There are two potential arguments that can be used to start a single node as shown above. If no coordinator IP:port is provided then it will be read from the properties file.

Additionally, you can start multiple nodes at once:

```
$ ./start-nodes-on-ports [ port1 ] [ port2 ] . . .
```

Using this start script you can start all processes in parallel without having to open n windows and start them individually. Just make you you provide (n-1) ports so you do not need to start any more nodes to satisfy the ready requirement.

The number of ports (arguments) entered will determine how many nodes will be started. If more ports are provided than necessary (more than n-1) then the nodes who were able to join first will continue as expected and only the “extras” will terminate execution.

For explicit examples of these, see [Figure 3.1](#).

In regards to “quitting”, if you control-C on the coordinator node, it will end all of the other nodes processes as well. Especially if you are using the multi-node script, this is the easiest and best way to guarantee that all processes have been killed.

3.1.4 Starting a Client

```
$ ./start-client
```

```
$ ./start-client [ ip:port ]
```

The client, like the coordinator, can be started without any arguments. It can also be started with the IP and port information about a node in the DFS so that it can immediately connect instead of querying the coordinator node for a random node.

3.2 Interacting with the DFS

Once the client has been started and has connected with a random node, the DFS's CLI will start up. This is the medium through which a user can interact with the system.

3.2.1 DFS CLI (via Client)

The CLI will provide you with the following instructions upon starting:



```

DFS CLI INSTRUCTIONS:
(Please make sure you comma-separate the commands and arguments)

WRITE, [filename], [contents to write to file]
> write, file1.txt, this will be the contents of the file
> write, file1.txt, -f pathFromRoot/fileContainingContents.txt

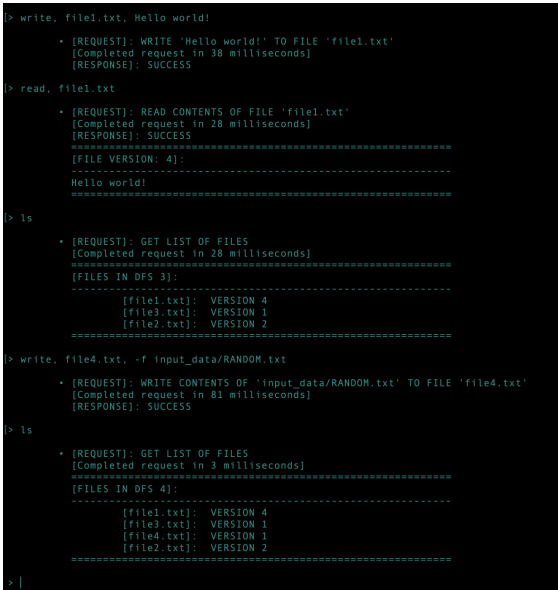
READ, [filename]
> read, file1.txt

LIST FILES AND VERSIONS:
> ls

SUBMIT FILE WITH MANY REQUESTS:
(each request on new line must be formatted as shown above (comma-separated))
> submit, pathFromRoot/fileWithRequests.txt

>
  
```

The user must enter one command per line and can continue to do so until they click enter without entering a command. Some simple examples are provided below:



```

> write, file1.txt, Hello world!
• [REQUEST]: WRITE 'Hello world!' TO FILE 'file1.txt'
  [Completed request in 38 milliseconds]
  [RESPONSE]: SUCCESS

> read, file1.txt
• [REQUEST]: READ CONTENTS OF FILE 'file1.txt'
  [Completed request in 28 milliseconds]
  [RESPONSE]: SUCCESS
  [FILE VERSION: 4]:
  =====
  Hello world!
  =====

> ls
• [REQUEST]: GET LIST OF FILES
  [Completed request in 28 milliseconds]
  [FILES IN DFS 3]:
  =====
  [file1.txt]: VERSION 4
  [file3.txt]: VERSION 1
  [file2.txt]: VERSION 2
  =====

> write, file4.txt, -f input_data/RANDOM.txt
• [REQUEST]: WRITE CONTENTS OF 'input_data/RANDOM.txt' TO FILE 'file4.txt'
  [Completed request in 81 milliseconds]
  [RESPONSE]: SUCCESS

> ls
• [REQUEST]: GET LIST OF FILES
  [Completed request in 3 milliseconds]
  [FILES IN DFS 4]:
  =====
  [file1.txt]: VERSION 4
  [file3.txt]: VERSION 1
  [file4.txt]: VERSION 1
  [file2.txt]: VERSION 2
  =====

> |
  
```

It is important that the contents of “write, filename, contents” does not contain any commas as the command itself is delimited by commas. If your contents contains commas, use the write from file option to do so.

For an example of how to use “submit”, please see [Section 4.2](#).

3.3 Cleaning and Resolving Issues

Due to my inexperience in writing bash scripts and catching interruptions to programs, I have also written some scripts to resolve issues caused by incorrect shut downs.

3.3.1 Cleaning

```
$ ./clean
```

This script does not accept any arguments, it merely runs different “cleaning” processes as needed such as removing *.class files, removing Thrift’s gen-java/ directory, and deleting the data/ directory.

For example, if the DFS is shut down while some node was writing to a file (which it could be doing due to a background update even if all user-entered requests are completed), then the coordinator script’s “clean” will be unable to remove that portion of the data directory. However, calling the clean script separately (after all nodes are shut down) will be able to properly remove the data directory.

Additionally, if the coordinator node did not shut down correctly, running the clean script will revert the properties file to the desired state so that it can be started again.

It should be called by the start scripts, but when things do not shut down properly that is not always the case.

3.3.2 Killing DFSNodeInstance(es)

```
$ ./kill-dfs-nodes
```

This script was written because I ran into some issues where ./start-nodes-on-ports would not always kill all nodes on interrupt (and wasn’t sure how else to resolve it due to my unfamiliarity with running parallel processes in bash), so this script kills all instances of DFSNodeInstance java processes.

It is also called at the end of the `./start-coordinator` script, which is why it is recommended to control-c on that process to kill ALL node processes (coordinator and regular).

3.3.3 Rebuilding Thrift Files

```
$ ./build-thrift
```

Lastly, this build script is called by the start scripts but was provided in case the user wants to do this process independently.

4 Testing and Results

4.1 Negative Tests

1. Starting a regular node or client before the coordinator has started:

```
[cse1-kh4240-02 (2): ~csci5105-pa3] ./start-node

=====
COMPILING NODE...
=====
Note: DFSNodeInstance.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
=====
STARTING NODE...
=====
[NodeInstance] EXPECTING AT LEAST 1 ARGUMENT, (0) PROVIDED.
=====
CLEANING UP...
=====
Removing thrift files...
Removing data directory...
Removing class files...

[cse1-kh4240-02 (4): ~csci5105-pa3] ./start-client

=====
COMPILING CLIENT...
=====
Note: Client.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
=====
STARTING CLIENT...
=====
[Client] Coordinator node is not yet live. Please make sure you started one and try again.
[cse1-kh4240-02 (5): ~csci5105-pa3]
```

2. Starting coordinator with invalid configurations

```

csc1-kh4248-02 (2): ~csc15105-pa3 ./start-coordinator n=3
=====
COMPILING NODE...
Note: DFSNodeInstance.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
=====
STARTING NODE...
(NodeInstance) Coordinator node configurations PRIOR to adjustment:
(COORDINATOR NODE CONFIGURATIONS):
-----
Running on csc1-kh4248-02.cselabs.umn.edu:9001
[QUORUM SELECTION]:    RANDOM
[N]:                   3
[Nw]:                  -1
[Nr]:                  -1
=====
(CoordinatorConfigurationManager) N entered (3), is less than minimum required number of replicas, changed to default (7)
(NodeInstance) Coordinator node configurations POST adjustment:
(COORDINATOR NODE CONFIGURATIONS):
-----
Running on csc1-kh4248-02.cselabs.umn.edu:9001
[QUORUM SELECTION]:    RANDOM
[N]:                   7
[Nw]:                   5
[Nr]:                   6
=====
(DFSNode)* Constructing COORDINATOR NODE on 'csc1-kh4248-02.cselabs.umn.edu:9001'
=====
(LIST OF NODES IN DFS (1 of 7)):
-----
[csc1-kh4248-02.cselabs.umn.edu:9001] *COORDINATOR
=====
SLF4J: The requested version 1.5.8 by your slf4j binding is not compatible with (1.6, 1.7)
SLF4J: See http://www.slf4j.org/codes.html#version_mismatch for further details.
(NodeInstance) Starting on 'csc1-kh4248-02.cselabs.umn.edu:9001'
(NodeInstance) INITIATED update #1
(DFSNode)* Received UPDATE() request.
(DFSNode)* File versions post UPDATE().
=====
(LIST OF FILES ON csc1-kh4248-02.cselabs.umn.edu:9001)
=====

csc1-kh4248-02 (3): ~csc15105-pa3 ./start-coordinator quorum_selection=USER_CONFIG
=====
COMPILING THRIFT FOR DFS...
Done compiling Thrift
=====
COMPILING NODE...
Note: DFSNodeInstance.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
=====
STARTING NODE...
(NodeInstance) Coordinator node configurations PRIOR to adjustment:
(COORDINATOR NODE CONFIGURATIONS):
-----
Running on csc1-kh4248-02.cselabs.umn.edu:9001
[QUORUM SELECTION]:    USER_CONFIG
[N]:                   7
[Nw]:                   5
[Nr]:                   -1
=====
(CoordinatorConfigurationManager) Invalid user entered Ns (1) and Nr (-1). Changing QuorumSelection from USER_CONFIG to RANDOM
(NodeInstance) Coordinator node configurations POST adjustment:
(COORDINATOR NODE CONFIGURATIONS):
-----
Running on csc1-kh4248-02.cselabs.umn.edu:9001
[QUORUM SELECTION]:    RANDOM
[N]:                   7
[Nw]:                   4
[Nr]:                   5
=====
(DFSNode)* Constructing COORDINATOR NODE on 'csc1-kh4248-02.cselabs.umn.edu:9001'
=====
(LIST OF NODES IN DFS (1 of 7)):
-----
[csc1-kh4248-02.cselabs.umn.edu:9001] *COORDINATOR
=====
SLF4J: The requested version 1.5.8 by your slf4j binding is not compatible with (1.6, 1.7)
SLF4J: See http://www.slf4j.org/codes.html#version_mismatch for further details.
(NodeInstance) Starting on 'csc1-kh4248-02.cselabs.umn.edu:9001'

```

3. Performing requests before DFS is ready

```

[> write, file1.txt, contents of file 1
• [REQUEST]: WRITE 'contents of file 1' TO FILE 'file1.txt'
  [Completed request in 46 milliseconds]
  [RESPONSE]: FAILURE (Coordinator not ready. Check all nodes have joined and try again.)

```

4. CLI Input Handling

```

[> jsdbfksj
[UNKNOWN COMMAND] Please try again.

[> read
[READ requires 1 argument] Please try again.

[> write, filename
[WRITE requires 2 arguments: filename, contents] Please try again.

> |

```

5. Submitting or requesting files that do not exist

```

[> write, file5.txt, -f this_file_doesnt_exist.txt
• [REQUEST]: WRITE CONTENTS OF 'this_file_doesnt_exist.txt' TO FILE 'file5.txt'
  [Completed request in 1 milliseconds]
  [RESPONSE]: FAILURE (Could not open file with contents to write.)

[> submit, this_one_doesnt_either.txt
• [REQUEST]: SUBMIT REQUESTS IN FILE 'this_one_doesnt_either.txt'
  [RESPONSE]: FAILURE (File 'this_one_doesnt_either.txt' not found)
  [Completed request in 1 milliseconds]

[> read, file_that_isnt_in_dfs.txt
• [REQUEST]: READ CONTENTS OF FILE 'file_that_isnt_in_dfs.txt'
  [Completed request in 4 milliseconds]
  [RESPONSE]: FAILURE (File does not exist yet)

> |

```


4.2 DFS Performance Tests

I performed one extensive performance test that compared the performance of the DFS with different **request distributions** (read-heavy, random, and write-heavy), different **quorum selection** settings (random, read_heavy, write_heavy, and consistent), and different **numbers of replicas** (7, 10, 15, and 20).

This was done by running each configuration 5 times, collecting the time required to complete the task, and plotting the results of the averages of each configuration. All components needed to re-create the test are provided.

To set up, I randomly generated three sets of 1000 requests on the same 10 files: READ_HEAVY.txt, RANDOM.txt, and WRITE_HEAVY.txt to 10 files. READ_HEAVY.txt is approximately 90% read requests and 10% write, RANDOM.txt is 50% and 50%, and WRITE_HEAVY.txt is 10% read and 90% write.

These files were generated by `src/tests/gen_test_files.py`. However, since they were randomly generated, the exact versions produced for the tests I ran were left in the input data directory if you would like to reference them.

Lastly, for the sake of fairness/consistency (and so tests can be re-run on the same DFS without having to re-start), an additional INIT_FILES.txt is provided that writes to each file once so that they all exist prior to running any of the following tests.

Then, for each configuration of the DFS (quorum configuration and number of replicas), the files were initialized by submitting the following to the client's CLI:

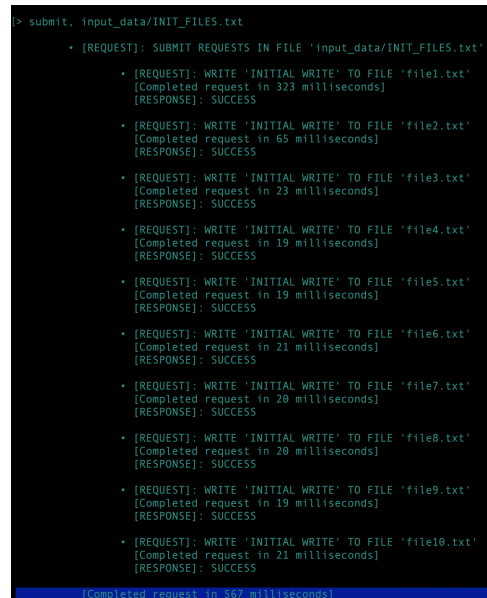
```
> submit, input_data/INIT_FILES.txt
```

Then, the current request-type's file was submitted (one of the following three):

```
> submit, input_data/READ_HEAVY.txt
> submit, input_data/RANDOM.txt
> submit, input_data/WRITE_HEAVY.txt
```

The time at the end of the request was recorded (see image to the right).

For all quorum_selection configurations except for RANDOM, all tests were run on the same instance of the DFS (without restarting it). To



```
> submit, input_data/INIT_FILES.txt
+ [REQUEST]: SUBMIT REQUESTS IN FILE 'input_data/INIT_FILES.txt'
+ [REQUEST]: WRITE 'INITIAL WRITE' TO FILE 'file1.txt'
  [Completed request in 323 milliseconds]
  [RESPONSE]: SUCCESS
+ [REQUEST]: WRITE 'INITIAL WRITE' TO FILE 'file2.txt'
  [Completed request in 65 milliseconds]
  [RESPONSE]: SUCCESS
+ [REQUEST]: WRITE 'INITIAL WRITE' TO FILE 'file3.txt'
  [Completed request in 23 milliseconds]
  [RESPONSE]: SUCCESS
+ [REQUEST]: WRITE 'INITIAL WRITE' TO FILE 'file4.txt'
  [Completed request in 19 milliseconds]
  [RESPONSE]: SUCCESS
+ [REQUEST]: WRITE 'INITIAL WRITE' TO FILE 'file5.txt'
  [Completed request in 19 milliseconds]
  [RESPONSE]: SUCCESS
+ [REQUEST]: WRITE 'INITIAL WRITE' TO FILE 'file6.txt'
  [Completed request in 21 milliseconds]
  [RESPONSE]: SUCCESS
+ [REQUEST]: WRITE 'INITIAL WRITE' TO FILE 'file7.txt'
  [Completed request in 20 milliseconds]
  [RESPONSE]: SUCCESS
+ [REQUEST]: WRITE 'INITIAL WRITE' TO FILE 'file8.txt'
  [Completed request in 20 milliseconds]
  [RESPONSE]: SUCCESS
+ [REQUEST]: WRITE 'INITIAL WRITE' TO FILE 'file9.txt'
  [Completed request in 19 milliseconds]
  [RESPONSE]: SUCCESS
+ [REQUEST]: WRITE 'INITIAL WRITE' TO FILE 'file10.txt'
  [Completed request in 21 milliseconds]
  [RESPONSE]: SUCCESS
[Completed request in 567 milliseconds]
```

average out the performances of different Nw/Nr distributions (when configured to RANDOM), the machine was re-started for each trial (but the same across all request types for each single trial).

All of the raw data as well as final plots of averages are provided below.

It was fun to see experimental confirmation that trying to achieve complete consistency will perform the slowest and that the READ_HEAVY and WRITE_HEAVY quorum selection configurations do actually perform the best for those kinds of requests.

RAW PERFORMANCE TEST DATA:

		Time elapsed (in milliseconds)											
		READ_HEAVY.txt REQUESTS				RANDOM.txt REQUESTS				WRITE_HEAVY.txt REQUESTS			
QUORUM SELECTION		N = 7	10	15	20	N = 7	10	15	20	N = 7	10	15	20
RANDOM (each trial was a random rw/nr distribution)	1	3522	5141	7178	8330	8445	15699	23521	30598	13197	22517	29517	50600
	2	3297	4844	5551	10417	6892	14113	13502	27279	12103	18528	20132	33701
	3	3646	4299	6586	10188	10139	11900	22109	27987	15593	14995	35018	37346
	4	3420	4998	5692	8715	7151	13763	18600	31382	11051	21287	32986	47371
	5	3845	5428	8205	7500	9080	13797	20069	28136	13468	21084	33323	38413
	AVG	3546	4942	6642.4	9030	8341.4	13854.4	19560.2	29076.4	13082.4	19682.2	30195.2	41486.2
READ_HEAVY	1	3546	3783	6081	6745	14465	15989	23216	32668	17474	22030	38662	52911
	2	3112	3796	5267	8433	12813	16387	23223	29289	18619	21795	39932	51774
	3	3008	3747	5225	6874	11764	14171	23632	31124	19978	23621	39147	50729
	4	3584	3762	5804	7310	11130	13848	28401	30105	18382	26634	39296	50549
	5	2362	3783	6014	6977	11687	14545	23991	30227	17271	23758	42247	52607
	AVG	3122.4	3774.2	5678.2	7267.8	12371.8	14988	24492.6	30682.6	18344.8	23567.6	39856.8	51714
WRITE_HEAVY	1	3297	4724	6412	8847	7523	11781	16806	24339	11020	17539	27583	38832
	2	3254	4164	5698	8706	8085	9866	15428	21358	11214	15803	22512	29615
	3	3051	4411	5859	8890	6549	10585	15296	20926	14022	15322	23226	30254
	4	3318	4210	5819	8004	6990	10689	16195	20603	11172	15543	22573	30240
	5	3317	8088	6330	9629	8505	10407	15553	21238	11685	16143	20941	28948
	AVG	3247.4	5119.4	6023.6	8815.2	7530.4	10665.6	15855.6	21692.8	11822.6	16070	23367	31577.8
CONSISTENT	1	4766	4532	8019	14880	11522	17867	29340	34664	19144	25710	39858	55235
	2	3633	5289	8970	13918	12469	17569	27812	32729	18217	27479	40277	54770
	3	3943	4958	7265	12106	12255	16359	26139	35574	18022	25361	40287	53970
	4	3549	5003	7519	12523	11840	15451	25847	32782	17589	23680	40133	52124
	5	7867	5008	8579	11257	11222	16542	26418	35171	20220	26064	42867	53522
	AVG	4751.6	4958	8070.4	12936.8	11861.6	16757.6	27111.2	34184	18638.4	25658.8	40684.4	53924.2

PLOTS OF AVERAGE TIME TO PERFORM 1000 REQUESTS (in milliseconds):

