Distributed File System Documentation and Performance Evaluation

Srisaipavan Valluri
vallu006@umn.edu

David Schoder
schod005@umn.edu

March 31, 2016

1 Abstract

In this project, we implemented a simple distributed file system which maintains consistency across all replicas by using a quorum-based approach. Without a consistency protocol, the file contents across all replicas would be out of sync and most clients wouldn't be reading the most up-to-date versions of the files. For example, it would be highly inconvienient if everyone saw a different ordering of replies to the same Facebook post. The quorum-based protocol offers resonable latency for read/write operations while preventing incredible network congestion. The implementation details, runtime procedure, and performance results are discussed below.

2 Components of System

2.1 Client Implementation

The implementation of the Client is located in **Client.cpp**. When run, the Client will initially retrieve a list of all the file servers that have registered with the Coordinator, including the Coordinator. Then it will display a menu in which the user can decide to write a file, read a file, display the list of file servers, or quit. If the user selects read or write, the Client will choose a random file server from the list and contact it. After the operation has completed, the Client will recieve the result of the operation. If the write failed due to voting, then the message, "Coordinator couldn't build write quorum" will appear. If the read failed, then "File does not exist" or "Coordinator couldn't build read quorum" will be outputted. If the user wants to read a file, then the version number will also be outputted.

2.2 File Server Implementation

The implementation of the File Server is located in **FileServerInterface_server.cpp**. When the File Server is run, it first registers itself with the coordinator and starts to listen for incoming connections. Here is the interface which is available to the Client:

- void Write(1:WriteFile fileInfo, 2:i64 flags)
- ReadFile Read(1:string filename, 2:i64 flags)
- list⟨ServerInfo⟩ getAllServers()

If you read through the FileServerInterface.thrift, you'll see that there are more functions in the interface. These will be explained in the Coordinator implementation (see below). If the File Server is not the Coordinator, then it will simply route the request to the Coordinator and wait until the Coordinator has processed the request. Once the results have returned, the File Server will forward the results back to the Client.

2.3 Coordinator Implementation

The implementation of the Coordinator is also located in the **FileServerInterface_server.cpp**. We felt that, since the Coordinator is a participating File Server, we could simply include a coordinator flag in the class declaration so that different segments of code would be executed in the same server stub depending on the coordinator flag. For example, in Write(), if the coordinator flag (specified at runtime) is 1, then the Coordinator code in Write() will be run. However, for any other File Server, the request will be forwarded to the Coordinator. This approach was taken for every interface function in order to distinguish between Coordinator and File Server responsibilities. Another advantage to this approach is that if the Client decided to directly contact the Coordinator, no code would need to be modified. There are other interface functions in the server stub, which aren't used by the Client. Instead, these serve as helper functions for the Coordinator to access information from the File Servers. These functions are:

- i64 Register(1:ServerInfo myInfo)
- i64 getVersion(1:string filename)
- WriteFile updateFile(1:WriteFile appendFile, 2:i64 version, 3:i64 append_flag)
- ReadFile getFile(1:string filename)
- list⟨FileInfo⟩ listFiles()

A queue, as mentioned in the specifications, is maintained by the Coordinator. When a Client performs a read or write operation, the request is forwarded to the Coordinator. This request is added to the queue and waits until it appears at the HEAD of the queue. As a result, this achieves sequential consistency because writes are not allowed to execute concurrently. For reads, if the next item in the queue is a read, then it will pop itself before executing. This allows other reads to execute concurrently. Afterwards, for each operation, the write or read quorums are constructed. If not enough File Servers are available or the requirements for N_w and N_r aren't met, then an error message is displayed at the Coordinator. This error message will also be displayed by the Client. For Write(), a thread is created so that any replicas not in the write quorum can be updated in the background. The Coordinator will also perform a **sync()** operation when the queue is empty. It will scan for the latest versions of all files and, for each file, will propogate the changes to every replica in the system. Since every sync operation causes network congestion, it's done periodically. However, the sync operation can be optimized. Instead of the Coordinator compiling the results and propogating updates, the Coordinator can send out a simple sync message to all of the File Servers. Each File Server is then responsible for updating its own set of files by contacting other File Servers. By distributing sync, it not only reduces the load on the Coordinator, but also decreases network congestion.

3 Running the DFS

Running the DFS must be done in this order. When running each component, it should be done on a different window.

3.1 Running the Coordinator

The Coordinator source code is located in thriftp2/gen-cpp/. Here are the commands to run the Coordinator:

```
cd gen-cpp

make all

./file_server 8456 1 <nr_number> <nw_number>
```

The Coordinator must run on VM x32-05 since the scripts use its IP Address and Port 8456 to generate File Servers and connect to the Coordinator.

3.2 Running the File Servers

Once the Coordinator is running, the File Servers must be run next. Here is the command to start 10 local File Servers, each assigned a different port number:

```
cd scripts

./startup_fileservers.csh
```

This will open 10 windows, each a different File Server. If you want to run a File Server remotely, this must be done manually. Here is the command line for a File Server if one wishes to run it remotely:

```
./file_server <port_number> 0 128.101.37.94 8456
```

3.3 Running the client

The client source code is located in thriftp2, which is the root directory of the project. For testing purposes, all test files must be uploaded atleast once with the command:

```
./client 128.101.37.94 8456 < ./scripts/client_upload_input
```

After this command, you may now run a single client to print out all the files in the system or manually read/writes files:

```
./client 128.101.37.94 8456
```

Or, you can run 4 clients concurrently by changing into the **scripts** directory and use any of these three commands:

- Read-heavy clients

  ```
  ./startup_clients.sh  client_heavyread_input
  ```

- Write-heavy clients

  ```
  ./startup_clients.sh  client_heavywriting_input
  ```

- Balanced clients

  ```
  ./startup_clients.sh  client_balanced_input
  ```

Each of these three commands introduces a different workload into the system, which was useful for compiling performance data.

4 Performance Results

To test the latency of read/write operations, combinations of N_w and N_r values were used under various workloads (read-heavy, write-heavy, and balanced clients). Each client was responsible for executing a batch of reads and writes.

4.1 Testing Procedure

All text files being used are stored in thriftp2/FILES. We tested 5 different pairs of read and write quorum sizes. These were (6,7), (5, 8), (3, 9), (1, 11), and (11, 11). For each N_w and N_r value, we ran the three different types of workloads. Thus, we were able to collect 30 data points from this experiment, which represent read and write times. Each data point is the average latency (ms) of a read/write operation under a certain workload with a specific N_w and N_r pair. Here are some tables of these values followed by a visualization:

Table 1: Average Latency (ms) of Read/Write operations for Balanced Clients

N_w	N_r	Read latency	Write latency
7	6	62.4	146.7
8	5	62.4	163.7
9	3	70.5	199.3
11	1	80.5	243.9
11	11	73.2	223.9

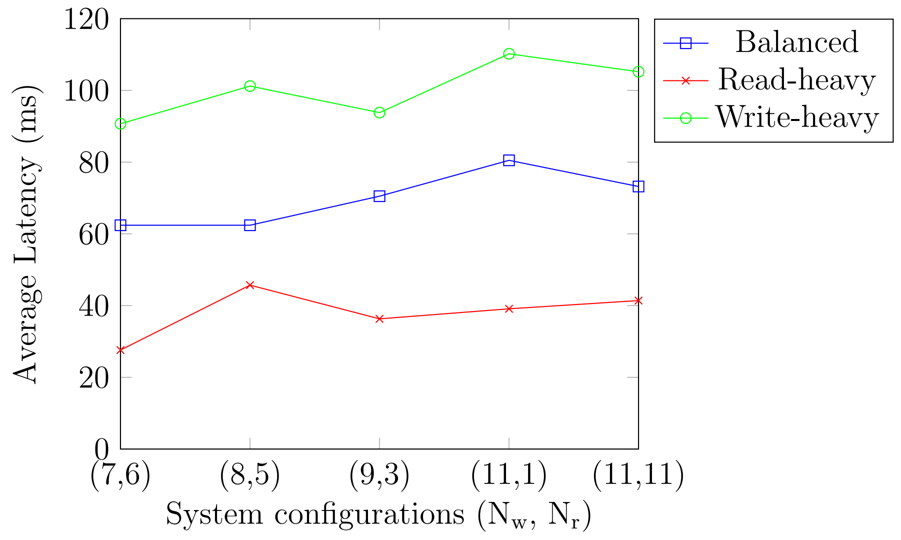
Table 2: Average Latency (ms) of Read/Write operations for Read-heavy Clients

N_w	N_r	Read latency	Write latency
7	6	27.6	134.3
8	5	45.7	171.9
9	3	36.3	161.8
11	1	39.1	187.1
11	11	41.4	191.5

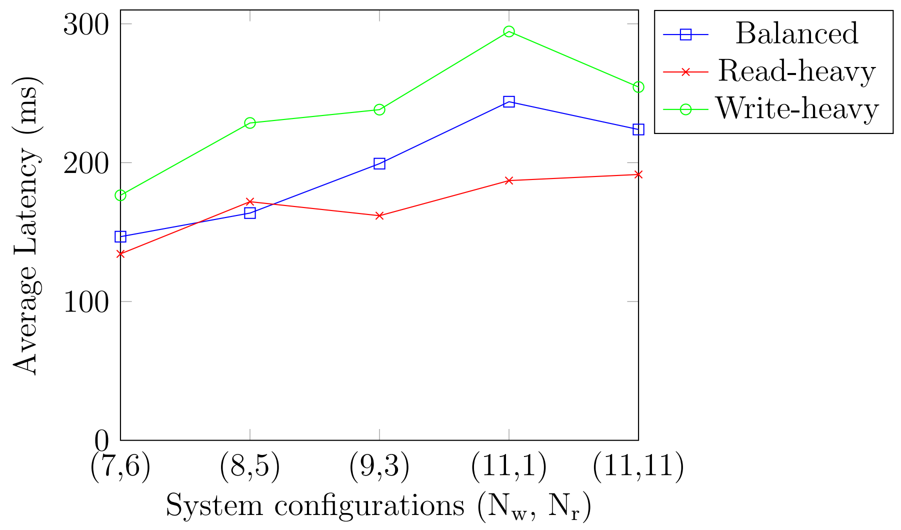
Table 3: Average Latency (ms) of Read/Write operations for Write-heavy Clients

N_w	N_r	Read latency	Write latency
7	6	90.7	176.5
8	5	101.2	228.6
9	3	93.8	238.2
11	1	110.2	294.5
11	11	105.2	254.5

Average Read Latencies (ms) under various workloads and system configurations



Average Write Latencies (ms) under various workloads and system configurations



Negative test cases: If N_w is greater than the total number of File Server running, including the Coordinator, then the Coordinator will print an error message and the write will fail. This will also happen when N_r is less than the total number of File Servers during a read. Another negative test case is when the client tries to write a file that doesn't exist in the FILES directory. This will print out an error message at the client-side.

4.2 Analysis

It can easily be seen that the read and write latencies are the highest for write-heavy clients for any system configuration. Also, the graphs exhibit a pattern in relation to the system configuration. As N_w increases and N_r decreases, the average latencies increase at a linear rate for both reads and writes. If the write quorum size increases, then there is increased overhead due to the number of File Servers that need to be traversed. This also affects reads because it increases the average wait time in the queue at the Coordinator. As a result, if the write latency increases, then the average read latency increases as well due to an increase in average wait time. The relationship between the balanced, heavy-write, and heavy-read workloads is clear. If we increase the number of writes, then other writes and reads will have to spend more time in the queue regardless of the system configuration. This is simply because writes are more expensive than reads. Naturally, for any system configuration, the write-heavy workloads have greater latencies than both balanced and read-heavy clients. Since balanced clients write more than read-heavy clients, they too have greater average latencies than read-heavy clients. In conclusion, different workloads affect the average latencies regardless of system configuration due to average wait time in the queue, which increases as the number of writes increases. This is one drawback to maintaining a stronger form of consistency.