Member:
Sihan Wei wei00114
Zongshun Zhang zhan4475

CSci 5105 - Introduction to Distributed Systems

PA3 Documentation

Overview

In this project, we have implemented a simple distributed file system(DFS), in which multiple clients can share files together. In this file system, the files will be replicated to several servers for increased performance and availability.
We implemented this DFS using the quorum based protocol.

Component Description

Coordinator

One of file servers will act as the control point for your quorum. A file server which gets a request from the client will contact the coordinator to carry out the operation. That is, any servers can receive read/write requests from users and they will forward the requests to the coordinator. The coordinator will then contact the other randomly chosen servers needed for the quorum to complete the operation.
In our DFS, the coordinator is chosen by us and also **functions as a server**.

Servers

There are servers containing replicas of the files. Randomly choosed servers will be contacted by the client at the beginning to forward requests to the coordinator. In our DFS, the default number of servers is 7(including the coordinator).

Client

There are multiple clients performing read and write operations on the shared files. The client can contact any server. In our DFS, the defult number of clients is 3.
Since the client has to request server and these two operations could not be completed at the same time. So, we divide our client into two parts, according to their functionality respectively.

(a) ClientReceiver

The ClientReceiver is used to listen results from the server with newest version and print the results on the screen. In a read operation, the content of the file and an “ACKR” mark would be printed. In a write operation, an “ACKW” mark and total number of operations sent by the client would be printed.

(b) ClientSender

The ClientSender is used to send requests to a random server. Since the client can contact any server at the beginning, we will choose a server when running the client.

Implementation Details

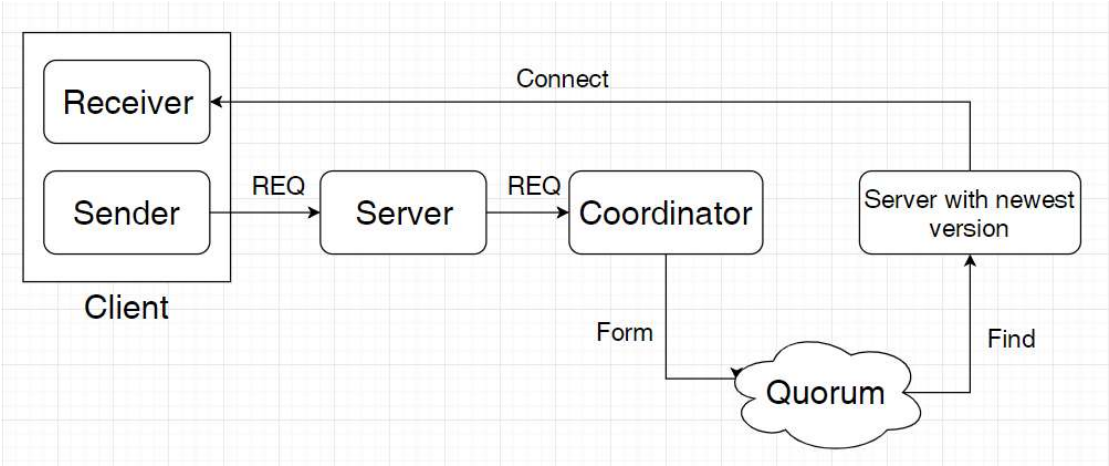
For a single file, the operations are sequential, thus write_back methods will be called sequentially, thus the results will be printed sequentially.

In our DFS, our read operations for a single file can be performed concurrently.

Whole process

In our DFS, the cilent first requests a server(read/write filename). Since in this step, the client can contact any server, we will at first specify a server(port) for the client to connect to. After the server has received the request, it will forward the request to the coordinator. Then, the coordinator will randomly choose N_R or N_W servers(including the coordinator itself) from an arraylist where it stores all the servers to form a read or write quorum. Then the coordinator would loop all servers in the quorum fo find the same file which the client requests with the newest version number. Finally, such server with newest version number would contact the client and print the content in the file on the client screen(if the request is read) or update the version number(if the request is write).

The whole process of an operation can be visualized as the following chart:



Concurrency Control

In our DFS, the operations, no matter writes or reads are performed concurrently for

different files. The write operations for a single file are synchronized. **We also implemented the concurrent read operations for a single file.**

Eventual Consistency

For the synch operations, we peridocially set flag to do synch instead of ExecReqs. This setter is an individual thread running in coordinator.
Because we always wait for all threads to finish in ***ExecReqs***, so we only need to check flag to decide whether to do synch.
Reader may notice our files under different servers got "synched" every 2000ms, we also printed the notification on coordinator screen.

File List

All the files stored on the server would be found in the corresponding directories(named as server IP and server Port) in the gen-java directory.

Setup and Execution

Compile

Firstly, you may need to compile the `.thrift` files to generate the `gen-java` directory. Then you may need to copy the `.java` files and test scripts into this directory. And finally, you need to compile all the `.java` files. You could simply run the following command to finish the whole process.

- bash ./make.sh

Execute

(a) First, change your current work path to `gen-java/`

- cd gen-java/

(b) Second, you need to start the Coordinator and Server.

- bash ./node.sh

(c) Third, you may need to start the ClientReceivers for the listening purpose.

- bash ./receiver.sh

Please notice that you may have to run ClientReceiver before running ClientSender.

(d) Finally, you may need to start the ClientSenders for send requests.

We provided 4 cases to test the performances of our system under different workloads. Here are the 4 workloads we tested and the corresponding commands you may use.

(i) Write-only

Under this circumstance, we prepared a script named `write-only.txt`, which is composed of 1000 write requests. To test with this case, you may using the following command:

- bash ./write-only.sh

(ii) Read-only

Under this circumstance, we prepared a script named `read-only.txt`, which is composed of 1000 read requests. To test with this case, you may using the following command:

- bash ./read-only.sh

(iii) Read-heavy

Under this circumstance, we prepared a script named `write-only.txt`, which is composed of 1000 operations. Among them, about 90% are reads and about 10% are writes. To test with this case, you may using the following command:

- bash ./read-heavy.sh

(iv) Write-heavy

Under this circumstance, we prepared a script named `write-only.txt`, which is composed of 1000 operations. Among them, about 10% are reads and about 90% are writes. To test with this case, you may using the following command:

- bash ./read-heavy.sh

(v) Custom test cases

You can also test our system with your own test cases. For this purpose, you could simply modify our `.sh` file. For example, you could modify `read-heavy.sh`.
The format of the input args of ClientSender is:

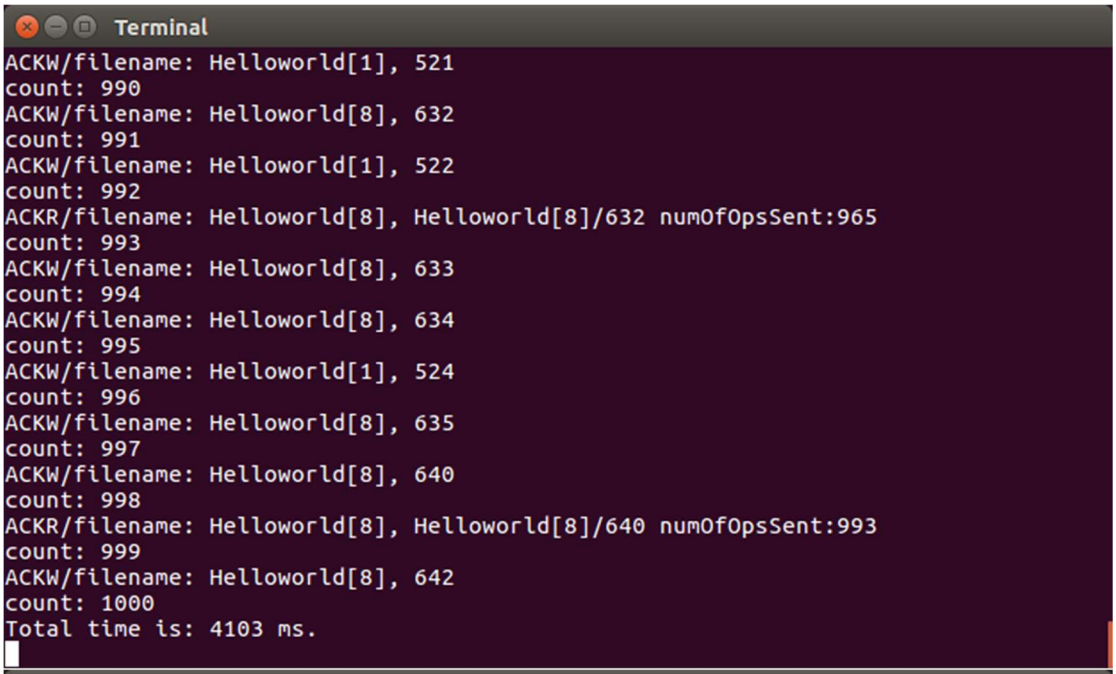
- ClientSender [serverIP] [serverPort] [receiverPort] [filePath]

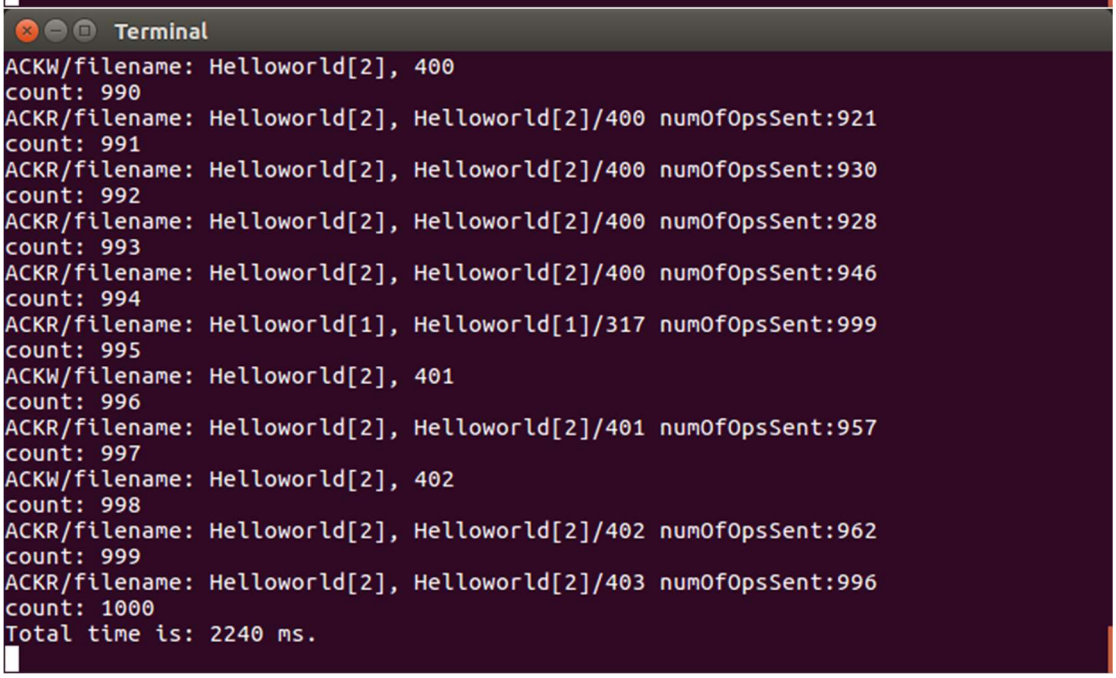
So you could simply **change the filePath parameter** in `read-heavy.sh`, which is `./read-heavy.txt`, **to your own file path**.

Please Notice that, you may also need to **change the serverIP parameter**, which is cs-exa in this case, **to the IP address of the machine you run the servers on**.

5. Testing Description

Positive Cases





We test our systems under four workloads:
write-only, read-only, read-heavy and write-heavy
and four (N_R, N_W) pairs:
(4, 4), (3,5), (2,6) and (1,7), where the total number of replicas is N = 7.
The performance is measured in time in milliseconds.

(i) (N_R, N_W) = (4, 4)

		(4, 4)				
		1	2	3	4	5
write-only	client1	7798	9006	7419	7753	7671

	client2	6765	7063	6775	6464	6707
	client3	7546	7960	7135	7186	7308
	AVE			7370.4		
read-only	client1	1520	1622	1643	1562	1658
	client2	1494	1569	1634	1526	1456
	client3	1501	1650	1639	1547	1618
	AVE			1575.933333		
read-heavy	client1	2264	2137	2689	2335	2210
	client2	2049	2073	2199	2217	2141
	client3	2236	2126	2321	2334	2207
	AVE			2235.866667		
write-heavy	client1	6786	5823	5531	5691	5644
	client2	5944	6507	6797	6925	6634
	client3	6409	6775	7182	7146	7037
	AVE			6455.4		

(ii) (N_R, N_W) = (3, 5)

(3,5)						
		1	2	3	4	5
write-only	client1	10812	8973	8914	8837	8411
	client2	10945	9737	9541	9568	9199
	client3	11116	10216	9907	10014	9388
	Ave			9705.2		
read-only	client1	1441	1500	1328	1391	1464
	client2	1460	1525	1428	1376	1466
	client3	1484	1474	1581	1358	1516
	AVE			1452.8		
read-heavy	client1	2150	2175	1889	1970	2034
	client2	2354	2300	1997	2085	2156
	client3	2729	2332	2024	2168	2223
	AVE			2172.4		
write-heavy	client1	6540	7028	7133	6919	6955
	client2	7414	7740	8138	7832	7728
	client3	8209	8069	8625	8865	8100
	AVE			7686.333333		

(iii) (N_R, N_W) = (2, 6)

(2,6)						
write-only	client1	12482	10420	9981	10587	9529
	client2	12526	11620	11880	11584	10967

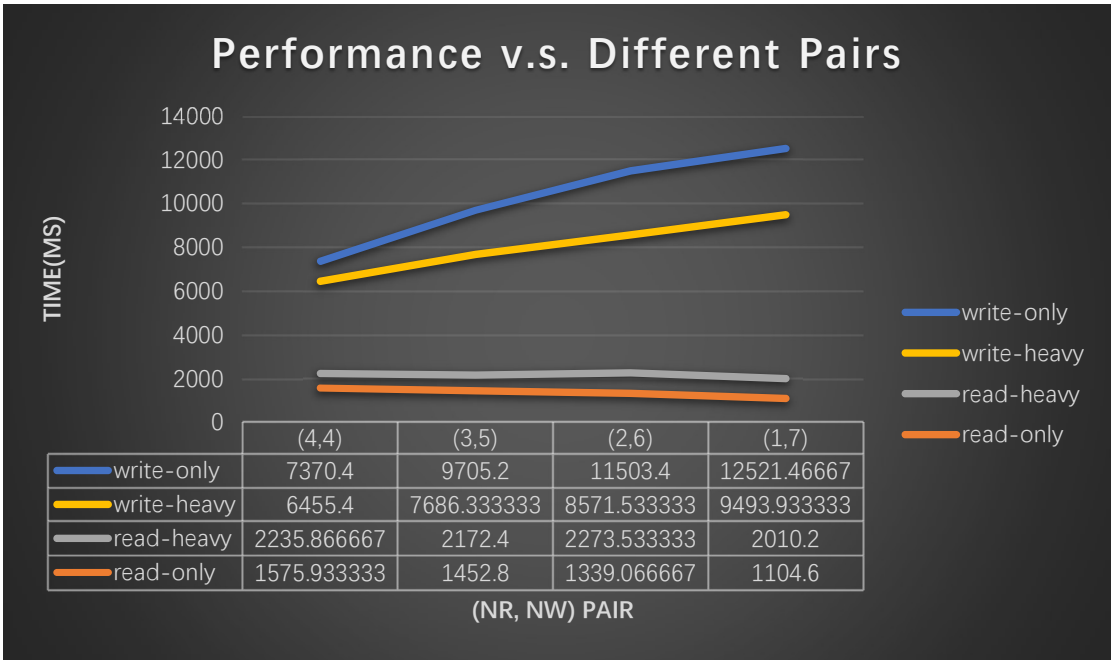
	client3	12879	12194	12314	12154	11434
	Ave			11503.4		
read-only	client1	1517	1250	1206	1340	1310
	client2	1514	1266	1216	1574	1277
	client3	1470	1183	1215	1549	1199
	AVE			1339.066667		
read-heavy	client1	2026	2178	2032	2346	2174
	client2	2158	2256	2157	2464	2286
	client3	2223	2341	2200	2615	2647
	AVE			2273.533333		
write-heavy	client1	8471	7476	7710	7719	7973
	client2	9043	8280	8674	8495	8682
	client3	9924	8707	9291	8994	9134
	AVE			8571.533333		

(iv) (N_R, N_W) = (1, 7)

(1, 7)						
write-only	client1	12673	11329	11479	10956	11346
	client2	13301	12978	12537	12275	12431
	client3	13502	13703	13247	12957	13108
	Ave			12521.46667		
read-only	client1	1133	1147	1228	1156	1027
	client2	1126	1159	1163	1094	1069
	client3	1083	1066	1067	1064	987
	AVE			1104.6		
read-heavy	client1	1804	1679	1985	2087	1726
	client2	2130	1857	2083	2220	1907
	client3	2339	1865	2076	2252	2143
	AVE			2010.2		
write-heavy	client1	8722	8437	8893	8658	8585
	client2	9723	9610	9687	9555	9442
	client3	10281	10177	10137	9982	10520
	AVE			9493.933333		

Performance Evaluation

We plotted the evaluated performance with respect to different pairs of (NR, NW) and the result is shown as the following:



	(4,4)	(3,5)	(2,6)	(1,7)
write-only	7370.4	9705.2	11503.4	12521.46667
write-heavy	6455.4	7686.333333	8571.533333	9493.933333
read-heavy	2235.866667	2172.4	2273.533333	2010.2
read-only	1575.933333	1452.8	1339.066667	1104.6

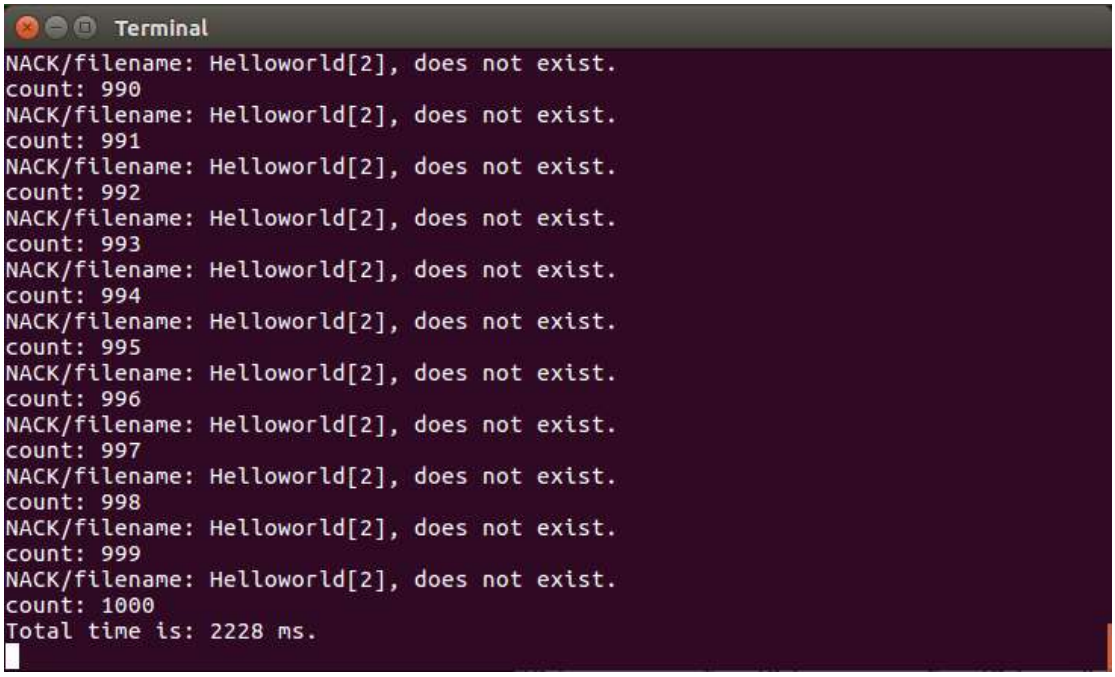
The total time for 1000 write operations(write-only) is about 7370.4 ms, which means a single write operation consumes about **7.3704** ms.
The total time for 1000 read operations(read-only) is about 1575.9 ms, which means a single read operation consumes about **1.5759** ms.
From this chart we can easily conclude that, the optimal (N_R, N_W) pair for a read-heavy wordload is (1, 7), where the **N_R is minimum**. This is because the fewer servers in a read quorum, the fewer read operations would be done while reading a file.
On the other hand, for a write-heavy workload, the optimal (N_R, N_W) pair is (4, 4), where the **Nw is minimum**. This is because the fewer servers in a write quorum, the fewer write operations would be done while writing a file.

Negative Cases

File does not exist

In terms of reading, we just reply “file not exist”.

In terms of writing, we create the file and ACK back client.


Terminal
NACK/filename: Helloworld[2], does not exist.
count: 990
NACK/filename: Helloworld[2], does not exist.
count: 991
NACK/filename: Helloworld[2], does not exist.
count: 992
NACK/filename: Helloworld[2], does not exist.
count: 993
NACK/filename: Helloworld[2], does not exist.
count: 994
NACK/filename: Helloworld[2], does not exist.
count: 995
NACK/filename: Helloworld[2], does not exist.
count: 996
NACK/filename: Helloworld[2], does not exist.
count: 997
NACK/filename: Helloworld[2], does not exist.
count: 998
NACK/filename: Helloworld[2], does not exist.
count: 999
NACK/filename: Helloworld[2], does not exist.
count: 1000
Total time is: 2228 ms.