**A3: fs533** **Lucas Palazzi, Jiahui Tang**
CPEN 533

# 1 System Design Overview

For this assignment, we want to build a simple distributed file system called fs533. Our system includes two parts: the failure detector and fs533 file system. For the failure detector, we reused our group membership program in Assignment 2. Similarly, a node needs to 'join' the group first in order to join fs533 file system. After joining the system, each node will keep a list of alive nodes by using the failure detector. This list will be used for the possible data re-update after a failure happened. For fs533 file system, to solve the challenges during horizontal scaling, especially the problems about data placement and partition management, we can either create a centralized system (with a node master) or decentralized system. In our fs533 system, we put our nodes and files into a virtual ring as a Distributed Hash Table (DHT), and used a decentralized approach to realize better scalability. Every VM node in the system can be considered as a master node of the system. Hence, if any node fails, we do not need to re-elect a new master node. Each node will maintain a file table which includes an id mapper and a file mapper.

- **Id mapper**: creates the mapping between vm_ids and files stored in fs533 system. The format is{*id1: (file1, file2...), id2:..., idn:...*} for a n-node system. When putting a new file into the system, every node will use modulo hashing (same hash function for all nodes: SHA256) to calculate which nodes the file should be placed on. When removing a file, we just simply delete the entries on the storage node. The entire id mapper will be printed when user input 'ls' in the terminal.

- **File mapper**: creates the mapping between files in fs533 and their details. Format is {*file1: {'status': 'wr-able', 'timestamp': now(), 'replicas': (id1, id2, id3, id4)} file2:*{...}, ...}. We keep three properties for each file: status, timestamp and replicas. 'Replicas' contains all the VMs that store replicas. In our system, we have four replicas for each file. 'Timestamp' is the time when the file was recently modified. 'Status' is either 'wr-able' (the file can be modified) or 'writing' (the file is being modified). We use 'timestamp' to deal with the problem that two put operations within one minute, and use 'status' to create a simple quorum (or simple mark) to prevent write-read conflicts.

***Replication strategy***: We implemented a data replication strategy similar to Chord (key's N successors or preference list). Since we need to tolerate two failures at a time, the size of our successors should be more than 2. In our system, we replicate data onto four nodes in the system. Each time we replicate an entire file. After a failure, we ensure that data is suitably replicated (re-update) to other alive nodes by looking up the alive list so that the system can tolerate other failures.

***Message Format***: We defined five types of message in our fs533 file system: JOIN, UPDATE, REMOVE, FAILED_RELAY, and STATUS-UPDATE. All the messages are transmitted as JSON payloads of UDP datagram. For JOIN messages, we only send the list of alive nodes. For UPDATE messages, they have five fields: type, filename, status, timestamp, and replicas. After a node receives an UPDATE message, it will update the corresponding information in its file table (id mapper and file mapper). REMOVE messages only have two fields: type and filename. The receiver will delete the entries after receiving a REMOVE message. Group Membership program will send a FAILED_REPLY message if it detected a timeout. The receiver will try to re-replicate the data after receiving a FAILED_REPLY message. STATUS-UPDATE

message is used to set the current status of a file. Since we also used Group membership program, so the network will also have all the other messages that we defined in assignment 2.

***Implementation***: In order to keep our code simple, we take advantage of 'scp' to read and write files into other nodes. Moreover, same as failure detector, fs533 file system uses UDP (port number 53314) to build communications between the VM nodes.

***Log Querying Tool:*** We save all the logs into a local file called fs533[hostname].log in each node, so we can use the log querying tool we built before to help us debugging and do the measurements. For debugging, we can get all the error messages in the log files.

***Unit tests:*** We tested our basic file operations such as 'put', 'get', 'remove' and 'ls' by generate some dummy files on the VM node being tested. The user can simply test our code by inputting 'test' after entering the file system. The test generates random files, stores them on the file system using 'put', then removes some using 'remove', and finally checks both the 'ls' listings and the 'get' contents to ensure the file system works properly. If all tests pass, the system will print 'Test passed' in the end.

# 2 Measurements

***Measurement 1: re-replication time and bandwidth upon a failure (you can measure for a 40 MB file)***

For the bandwidth measurements, we used *iptop* to help use get all the results. For a 40 MB file: Average re-replication time is 0.157s (standard deviation: 0.013s). Average Nandwidth upon a failure is 47.16 kb/s (standard deviation: 1.16 kb/s). All the results are based on 5 different trials and the measurement point is the first node that stores the file. Since in our system, we keep checking the status of nodes. If a node receives a FAILED_REPLY message, it will start replication process immediately, so the average re-replication time is rather short.

***Measurement 2: times to insert, read, and update, files of size 25 MB, 100 MB (6 total data points), under no failure***

All the results are shown in the table below and Figure 1. As we can see from the results, operations on larger files will take more times because we used 'scp' to help use to transfer the file. For the 'insert' and 'update' operations, the times are almost the same. This is because we replicate the entire file instead of just updating the new part. It is also noticed that the standard deviations are small, it is because the file transfer performance is very stable in the system. 'Read' operation only takes up only about a quarter of the time of 'insert' operation.

| | ***Insert*** | | ***Read*** | | ***Update*** | |
|---|---|---|---|---|---|---|
| ***Filesize*** | **25MB** | **100MB** | **25MB** | **100MB** | **25MB** | **100MB** |
| ***Average time (s)*** | 7.1096 | 10.367 | 1.617 | 2.0592 | 4.5202 | 8.6856 |
| ***Stdev*** | 0.0685 | 0.229 | 0.2756 | 0.0854 | 0.2503 | 0.2042 |

***Measurement 3: Time to detect write-write conflicts for two consecutive writes within 1 minute to the same file***

Average time to detect write-write conflicts for two consecutive writes within 1 minute to the same file is 1.034ms (standard deviation: 0.108ms). Since for each file, we store the timestamp when the file has been recently modified, so we can easily detect two writes within 1 minute.

***Measurement 4: time to store the entire English Wikipedia corpus into fs533 with 4 machines and 8 machines (not counting the master).***

The results are shown in the table below and Figure 2. Since we used four replicas storage strategy, putting file into fs533 system is not related to the number of machines in the network

as long as the total VM nodes is larger than 4. Hence, as we can see from the results, the times are almost the same for 4 machines and 8 machines.

| | ***4 Machines*** | ***8 Machines*** |
|---|---|---|
| ***Average time (s)*** | 94.5774 | 97.772 |
| ***Stdev*** | 1.1841 | 3.9867 |

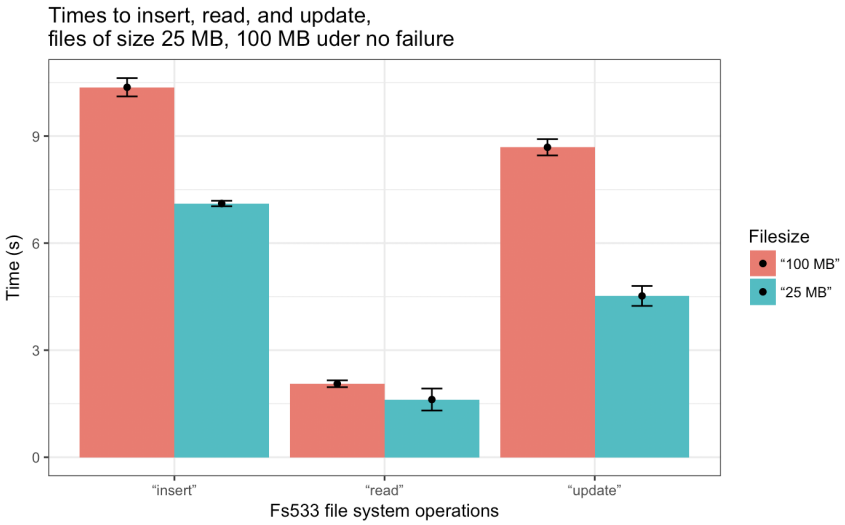


Figure 1: Measurement 2: Operation times for files of size 25MB, 100MB

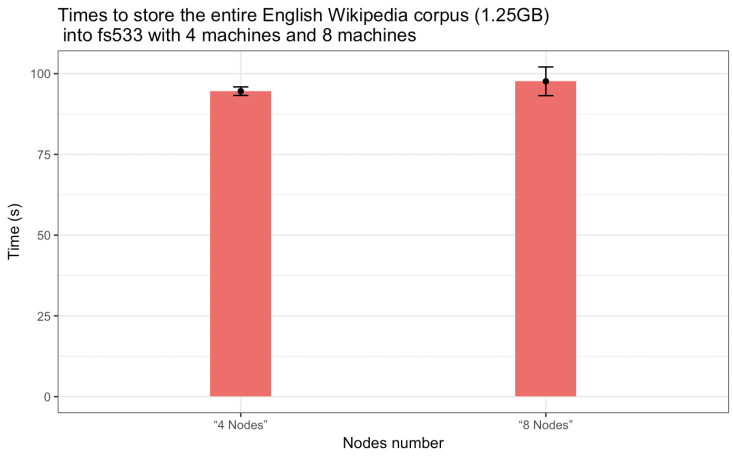


Figure 2: Measurement 4: Times to store the entire English Wikipedia (1.25GB) into fs533.