CSCI - 5105 Introduction to Distributed Systems
Project 3: Simple xFS - Peer to Peer Serverless File system

Rohit Sindhu [**sindh010**], Aravind Alagiri Ramkumar [**alagi005**], Dhruv Kumar [**kumar434**]

---

*[See **README.md** in the code folder for instructions on running the code and test cases.]*

Notes:- We have used TCP, Java to implement the Serverless file system and MD5 hash for the file checksums.

Assumptions:- The files is considered as immutable. They can be created and deleted but not updated.

System Implementation

***SERVER*:** *Maintains a hashmap of file name to list of peers containing file and another to store the checksum of each file.*

*** Find Operation:** Find is called with file name and server responds with list of peers hosting the file and the checksum.

*** UpdateList Operation:** When peer calls update list with list of file names and checksums, server updates it's hashmaps. The server adds the peer to the file mapping list if the file is already present or else will create a new one. Server also makes sure to delete the peer info from files which not present in the current list of files. Along with it, the checksum for that file is also updated in the respective hashmap if it is not already available.

*** Send Info Operation:** The server on start sends SEND_INFO request to all peers to fetch the peer's list of shared files. This is used to create the soft state of server and as fault tolerance mechanism for when server recovers after a crash.

***PEER*:**

The peer takes the shared directory path and port as the arguments and makes sure that the directory is created if not present and then creates the latency mapping to all other peers. The client starts the DirectoryUpdateThread which checks for file additions/removals in the shared directory for every specified time interval. The DirectoryUpdateThread will call the server's update list function **only if** there are changes present.

*** Find operation:** The client requests the server for the list of peers hosting the file and checksum by sending a file name.

*** Update List Operation:** The client creates a list of files present in the shared directory along with the checksum and sends it to the server for maintaining the correct state.

*** Download File Operation:** When a peer wants to download a file, it sends a find request to server and gets the list of the peers which has the file. Then it gets the load on all the peers in the list received from the server. Once it gets the load using the algorithm explained below, it inserts into a priority queue sorted in the increasing order of the load. We will process the peers one by one until we are able to download the file or until all the peers in the priority queue are checked. We take the best peer from the queue and try downloading, if it doesn't work, it chooses the next best peer from the queue. If the file is not able to be downloaded from any of the peers, the function returns a file not found error.

*** Send Info Operation:** The client on receiving a SEND_INFO request from server, send the directory file and checksum information to server.

*** Blocking find and update list operation on server crash:** During the find and update list operations, if the server crashes in between, we set a flag per peer indicating that the server is down and the peer waits on the flag for the server to come up again. At the same time, we start a singleton ServerHeartBeatListerner thread which pings the servers at regular intervals to check whether the server up again. If the servers responds to the ping, the thread will set the flag as false indicating that the server is up. The peer waiting on this flag will now start executing the process normally.

***FAULT TOLERANCE MECHANISMS*:**

*** Handling checksum validation failure:** If the checksum validation is failed while downloading, the client will retry for a specified number of times using the same peer before selecting the next best peer.

*** Handling server crash and recovery:** Whenever the server starts/recovers from a crash, the servers sends a SEND_INFO request to all the peers. Once the peers responds with the list of shared files and checksums, the server updates its respect hash maps and hence creates it soft state.

*** Handling peer crashes:** Whenever the peer starts/recover from a crash, the peer sends its update list operation to the server.

*** Special cases handling:** If all the peers in find list are down or file is not found on all peer, we return file not found to the client/peer which is trying to download the file. The design principles we followed are Leave it to the Client and make it fast. We thought, in our system that the operations should be fast and should not be waiting non-deterministically. So even if it there is an error, we should return it the client and let the client makes the choice whether to retry it if needed.

ALGORITHM FOR GET_LOAD: [*USED AS COST MEASURE IN PEER SELECTION*]

Our algorithm incorporates the following factors while selecting a peer:

1. **Number of physical cores at the peer:** Higher the number of physical cores, higher the number of concurrent requests which can be served by the peer without loss of performance.
2. **Number of uploads (N_u):** Higher the number of concurrent uploads, higher the waiting / response time for a new download / upload request.
3. **Number of downloads (N_d):** Higher the number of concurrent downloads, higher the waiting time for a new download / upload request.
4. **Load Factor (α)**: This is defined as the ratio of the total number of uploads and downloads with the number of cores(N_c). Note that this is different from the load index mentioned in the assignment.
5. **Average file size per upload/download($FS_{average}$):** Higher the average file size per upload/download, higher the waiting time for a new download request.
6. **Network latency between the peer and the requesting peer(L_s):** Higher the network latency (low bandwidth), higher the serving time for a particular download request.

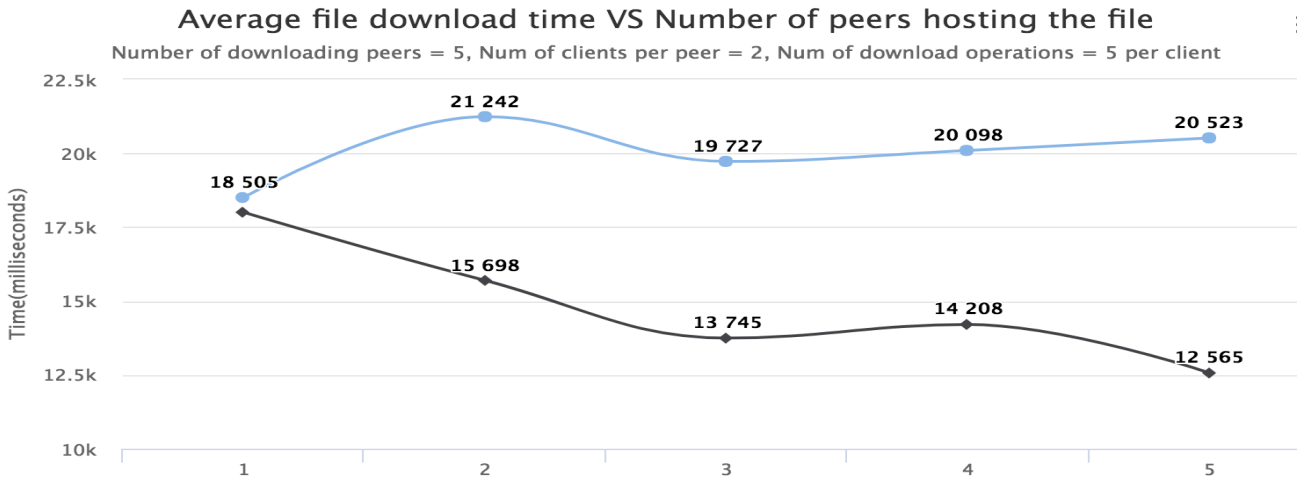
Combining all the above factors we get cost function where T_{total} provides the total cost.

$$\alpha = \frac{N_u + N_d}{N_c} \qquad FS_{average} = \frac{\sum_{i=0}^{N_u} U_i + \sum_{i=0}^{N_d} D_i}{N_u + N_d} \qquad T_{total} = (\alpha * FS_{average} * T_{processing}) + (L_S * FS)$$

U_i is the size of ith file being uploaded by the peer for other peers to download. D_i is the size of ith file being downloaded by the peer from other peers. $T_{processing}$ is the time taken by the peer in processing a file of unit size. FS is the size of file requested for download. We compute T_{total} for all the peers under consideration and then select the peer which has the least T_{total}

Performance Analysis:

Naive Algorithm gets the list of peers from the servers and choose any one of the peer from the list irrespective of the load



The basic analysis is that in the naive algorithm, even though the number of peers of hosting the file increases, the average time to download might not reduce as the peer selection strategy is very naive such that it selects the very first node in the list irrespective of the load, latency and replication. Since all the peers downloading a file get the same list order, most of the requests will go to very less number of servers.

But in our algorithm, we make sure that we route the requests and balance the load evenly according to the current load and latency of the peers. Hence, it can be seen the average time taken to download the file decreases as the number of peers serving the peer increases because we balance the load across peers.