CSCI 5105 Project 3: xFs System

Group members:
Shengping Wu wu000376
Fushan Wang wang9550

A. Instructions explaining

Client (**Node**):

1) Compile: go into Client/ directory and run `make`
2) Run `./Client.out [host][nodeID]` ,where host is the registry of the coordinator's address (This will allow client start with contacting the coordinator server and get the full server list back for further interaction), and the nodeID is the ID you would like to grant for this node. This ID will be used to identify nodes and read latency files.
3) User can type the command as prompted:
   - `download` will ask the user for the filename, and then it will download this file from other nodes.
   - `getload` will ask the user for the node number, and then it will return and display the load number of the input node.
   - `getload` will update the local file list to the tracking server.
   - `find` will ask the user for the filename, and then it will return the information of all the nodes that contains this file, in the format of `[nodeID;nodeIP;...]`
   - `debug` will ask the user for a filename, and then it will emulate the case that all the available nodes start to download this file to their local storage simultaneously.
   - `exit` will exit the current node.

Server (**Tracking Server**):

1) Compile: go into Server/ directory and run `make`
2) Run `./Server.out` to start the server. The Server should be started **prior to all the nodes** at the beginning, because every node will rely on the address of the tracking server to start.

B. Design Detail

1. Node - Download Algorithm
   I. A node could download files from other nodes, and also become the source of file for other download requests.
   II. When a node attempts to download a file from other nodes, it will first call Find() to the tracking server. The tracking server will return the information (`[nodeID;nodeIP;...]`) of all the nodes that contain this file. After receiving and loading this information, the node will firstly find the node with **minimum load**.
   III. The current node will load the information of load along with the latency information (which is stored globally as *latency.txt*) into a **Graph** that in which, the nodes of graph is composed of the nodes that is available, and the path between nodes in graph are the latency between the 2 nodes. This graph will be used to find the path that has the minimum path from a source node to a target node.

IV. If there are multiple nodes that have the minimum load globally, the node will **check the shortest latency path** from the current node to the minimum load node. The current node will pick the node that has the minimum latency path length. If there is only 1 node that has the minimum latency, the current node will directly find the shortest latency path to this target node.

V. After finding the shortest path, the current node will start to request along this path for downloading the designated file. This will form a **forward-download chain** that stretches from the first node to the target node. (For example, if the found shortest latency path is node 1 -> node 3 -> node 2, the node 1 will request node 3 for the target file, and the node 3 will in turn block this request and ask node 2 for the target file. After node 2 has obtained the designated file, it will send the file to the node 3, which in turn will send the file to node 1.)

VI. After receiving the designated file, the originated node will **check the file completeness** first. If everything is okay, the target file is downloaded. Otherwise, there might be 2 cases:

VII. For the first case, the downloaded file could be *corrupted* and the checksum validation would fail. In this case, the current node will try to download from the same path to fetch the designated file.

VIII. For the second case, the forward-download chain could be interrupted by a failing node transmission or a failing node. In this case, the current node will try to **directly contact** the designated node by its address. If the direct connect to the designated node also fails, this indicates that the target node is failed and can not respond to the download request. In this case, the current node will switch to find the next-minimum load node that contains the target file, and repeat the steps above.

IX. If the current node has gone through all the available nodes but still can not find the target file, we could infer that this file is **globally unavailable** at this time. The download will cease as a result.

2. Node - Other Operations

I. **Heartbeat**: the Heartbeat operation is a ***periodic ping*** operation that originated from every node to the tracking server. Every 5 secs, the current node will try to connect with the tracking server to make sure the tracking server is still alive. If so, the tracking server will return a SUCCESS flag; Else, when notice that the tracking server fails to respond in the maximum timeout, the node will regard the tracking server as fail and try to reconnect to the tracking server. When the tracking server goes back online, the heartbeat operation will successfully connect with the tracking server. After that, everybody will start an **updatelist** operation to the tracking server, therefore the tracking server will have all the file information of all nodes on returning back online. Every node will have a thread to separately continuously conduct the HEARTBEAT operation.

II. **Debug**: this is the operation specially designed for testing the multiple download cases. The debug operation will send a request to every other node that makes them do a Download operation at their local node. After requesting every other node doing so, the current node will also start multiple Download operations, and we will measure the average download time for the performance report. Everybody will download the same file at their local node as a result. Therefore, every node will have a thread to separately listen to the DEBUG operation.

3. Tracking server

   There are three functional threads in the tracking server:

   1) `FindRes_Getload()` to respond `Find()` in nodes by returning the list of nodes which store a certain file.
   2) `RecvList()` to receive `UpdateList()` from nodes to store the information about the nodes and files.
   3) `Heartbeat()` to allow nodes to check out the normality of the tracking server.

4. Fault Tolerance

   1) Downloaded files can be corrupted. In this case, we just restart the download process again to re-fetch the file from the original peer. To emulate the corrupted case, we allow the transmitted data to be **20%** percent corrupted (i.e., file content adds an extra byte) during the transmission.
   2) Tracking server crashes. We detect this by peers connecting to the tracking server periodically in `Heartbeat` thread, once the connection fails, all the peers will know the tracking server crashes. Once the tracking server rejoins the network, all the peers will send `UpdateList()` to assist the server to recover the file sharing information.
   3) Peer crashes. In this case, the receiver node will be time out in receiving files. There are two cases that can happen: 1) some intermediate nodes in the route crash, so we try to directly connect to the sender node first. If we cannot connect to it, it means 2) The sender node can be down. In this case, the receiver node can try to download from another node which has the next lowest cost.

C. Test Case

1. Performance graphs

   For the test case, we use the debug to create the situation that every node is trying to download some files. And for the measured node, we measure the average time that this node downloads a target file. A larger perspective is that all the nodes are initiated with dealing with the assumption that the nodes groups contains 10 nodes in total, so that for the case that the number of nodes is less than 10, there might be the case that a node might be regarded as unavailable if some nodes are trying to download file from that node.

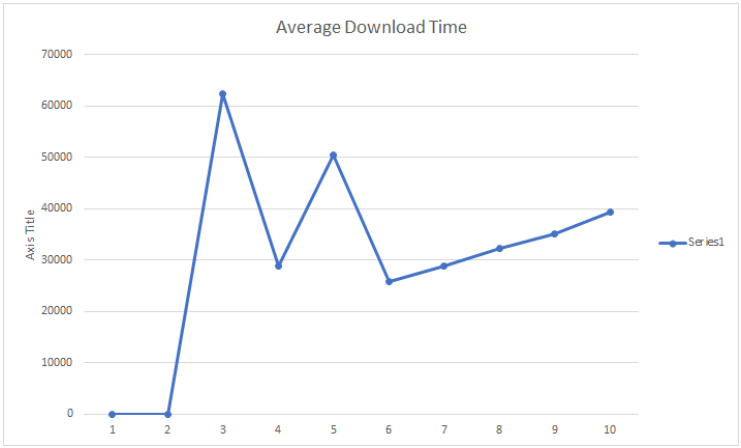

Fig 1. response graph

From the graph, we could tell that the response time has 2 stages with respect to the increasing of the nodes. The horizontal axis refers to the number of available nodes (therefore, we ignore that case that only have 1 and 2 nodes), and the vertical axis refers to the response time.

For the first stage of response time (3 nodes to 6 nodes), the response time is swing and unstable. This might because most nodes of all assumpted nodes (i.e., 10 nodes) are unavailable, and thus the measured node will be more likely to meet with the failing node case and therefore has to use the algorithm to change downloading strategy, which takes extra time.

For the second stage of response time (6 nodes to 10 nodes), the response time is slowly increasing with respect to the number of nodes. This might because in the situation that most nodes are available and less fault tolerance cases will be dealt with, the major factor that affect the download time would be the number of nodes, since with more node requesting for downloading, more concurrent downloading and uploading are common among all the nodes, therefore the average response for a single download operation would increase.

2. Fault Tolerance

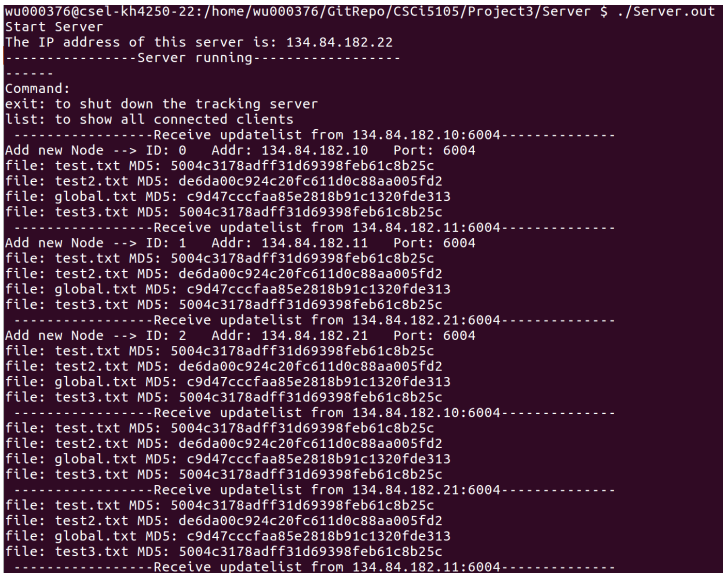

Fig 2. Tracking server UI and output

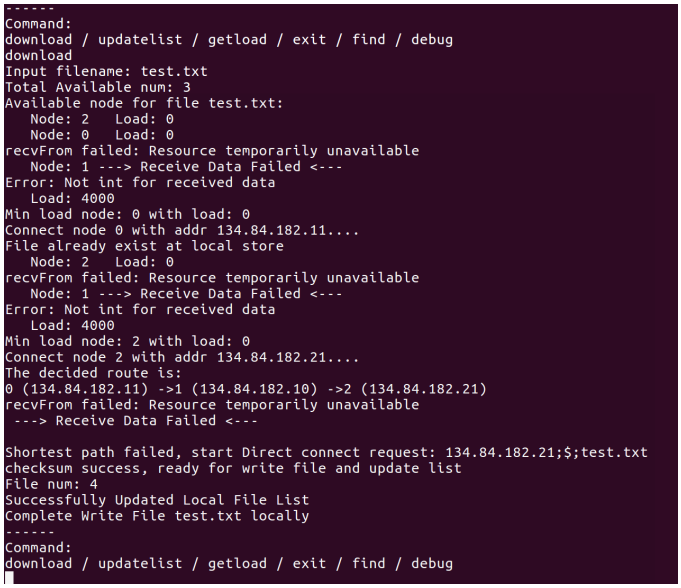

Fig 3. Intermediate peer crashes,receiver node try connect directly

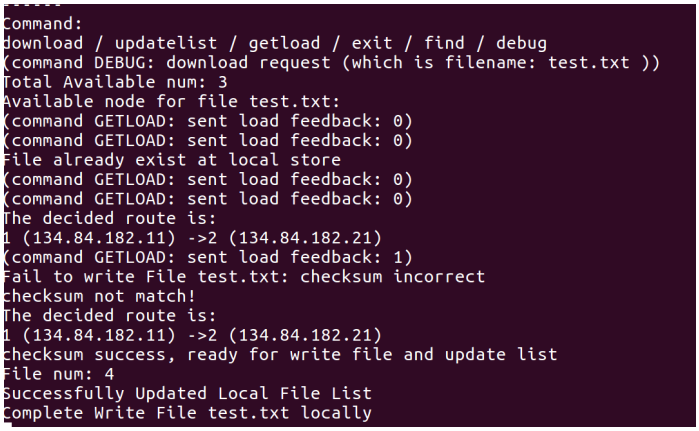

Fig 4. Downloaded file is corrupted (checksum incorrect)

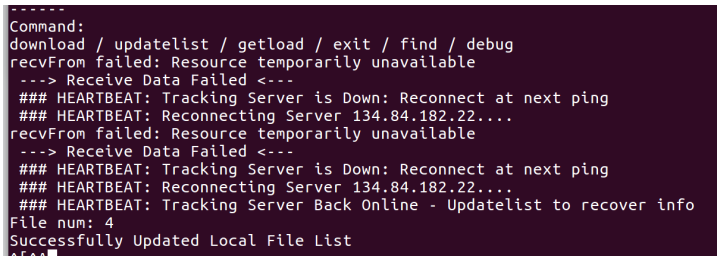

Fig 5. Tracking server crashes and recovers

D. Known Limits

- Since we use stoi() when loading the latency.txt, so the latency should not exceed 3500, or it will cause “std::out_of_range” error.
- Since all peers share the same files under the same account in CSE lab, all the nodes have the same initial files. In order to test, we ignore the current node in the `nodeInfoList` when downloading. This will result in the minimum path just directly connecting to its neighbor because every node has the same initial file copy. For the same reason, when the target file exists locally, we decide to let the node ignore this local file and try to find this file at other nodes in order to get a better test result.
- Source code: https://github.com/Inutto/CSCi5105/tree/master/Project3