DESIGN DOCUMENT

CS550 - Advanced OS

Fall 2020

Amit Nikam

A20470263

anikam@hawk.iit.edu

Implementation

Overview: In this assignment I have implemented a P2P architectured file system where the node can download files at chunk level from other nodes in the network. A node can become a Leader which has a Distributed Hash Table, which keeps a record of all the files and correspond nodes in the network and also a second list of nodes who had intended to download one of those files. Both these file lists contain the address of the corresponding nodes. All the nodes within the network are in touch with other active nodes in the network dynamically and keep a record of the leader. The nodes are multithreaded and can connect to multiple other nodes at the same time. Nodes are capable of uploading and downloading files in parallel. Infact nodes are capable of uploading or downloading specific fixed sized chunks of the file as and when required parallaly. Nodes use sockets along with a set of predefined protocol to establish connection and communicate. Sockets are software structures within a network node of a computer network that serve as an endpoint for sending and receiving data across the network. This program is implemented using python 3.8.3 and it's default modules.

Node: Upon initialization the node scans the network i.e. ports 9000 to 9130 to look for other nodes. Incase no other nodes are found the node elects itself as the Leader. If there are other nodes in the network, the node pings all other nodes to see who the leader is. Upon finding the leader in the network the node updates it's memory of the leader and synchronizes with the leader by updating the list of files it has on the DHT. Incase there were no leaders amongst the nodes in the network, the node elects itself as the leader and informs other nodes of the same. All other nodes there after send their list of files to the DHT. All of these communications are considered as synchronization or DHT communications and come under **leader handler.**

The node implementation happens to be a server and client at the same time. Depending upon the case, the node can become a server or be a client to other nodes. While initializing the node, the user can pass arguments to declare where to host the files for the node, set node IP address or set node port to listen for connections. If the file directory is not created, the node creates one. Each node has a unique file and log directory with it's port number.

The DHT node has no limit on maximum connections at a time. Thus any number of nodes that are joining the network can try and connect to DHT Node. After reaching the maximum capacity, if the nodes request DHT for service, they end up calling for the election again and a new leader is elected. The node establishes connection with the dht node as a client since it is trying to get the file list and the source list. The node fetches the list and the user selects the file to download. The node requests the user to provide some input for selecting the file they want to download. Then the nodes fetches the corresponsing source lists for the file. Both confirmed list and list of nodes that might have the file are fetched from

the dht, and both contain the addresses of the nodes who can serve the file.

The node then take some sort of user input to decide which sources to download file from. After the source list is ready, it establish a threaded connection to one of the sources from the confirmed file host list and fetches the meta data for the file. These connection related to downloading of file come under **a download handler** which is a function to manage the proper downloads of file chunks from different sources.

Within the download handler, once the file's meta data is fetched and sources are decided, the handler uses a windowed round robin algorithm to decide which chunks to download from which available sources. In this the total chunks of the file are divided by the number of sources selected and their floor is taken to get the window(set of chunks), say W_S. Then next modulo of the total chunks divide by total sources is taken, say M_S.

So we have

W_S = floor(#chunks in a file / #selected sources)

and,

M_S = #chunks in file % #selected sources.

Such that,

(#chunks in file) = W_S * (#selected sources) + M_S

Now we simple map these windows(set of chunks) to the sources and add the chunks form the M_S to the first source since in round robin we loop around the elements. Since this is windowed round robin, each next window is assigned to next source, irrespective of the number of chunks within because M_S is always smaller than W_S. If M_S were to become equal to W_S it would get equally split among the sources as total number of selected sources would be divisible by W_S.

Example: If a file of say 11 chunks was to be hosted by 3 sources. The client node would get a W_S = floor(11/3) = 3 and M_S = 11%3 = 2. So the three sources, lets say A,B and C would send 5(1 - 5), 3(6 - 8), 3(9 - 11) file chunks respectively. If it were 12 chunks, each window would be 4,4,4.

The listener thread on node acts as a server and is listening on the specified port. It serves the other nodes which are looking to download chunks of files. The node listener is a request-response type thread and so it serves the clients on demand. It is **a client handler.**

Each and every node in the network has a leader handler and a client handler, but a node may have a download handler only if it is downloading files from other nodes. All these 3 handlers together make up the Node.

Protocol: The nodes follow a strict protocol for file chunk size which is 1536 Bytes of file data. This size is picked with the idea that the data chunk would be wrapped with meta data related to the file, chunk and transfers which will bring it close to 2000 Bytes size. Since each payload in the network happens to be 2048 Bytes, this would fit chunks in a payload perfectly. The DHT server and nodes follow certain communication protocol. For every message, a header of 16 Bytes is padded at the

beginning of the message. This header contains the length of the payload. The header is first decoded and validated before accepting the rest of the message. Messages have a packet size of 2048 Bytes, so any messages larger then that are sent across in multiple packets. A default timeout of has been set for messages so that any packet loss shall terminate the message. All the messages are marshalled and unmarshalled using python's pickle module, which happens to be a serializer. Serializing the messages enabled us to send python lists, data chunks and objects through the message payload.

Possible Improvements

One of the major area of improvement in my opinion happens to be in the way a file is stored. Currently each file is stored entirely in some of the nodes. Instead, it would have been much better to store chunks of files across different nodes with meta data available at the DHT.

This way multiple chunks could ensure replication and less changes of failure. In current implementation, assuming if two nodes have a file and if both fail, file becomes unreachable. But if this was spread across the network in chunks, the chances of failure would be reduced significantly as somewhere another backup chunk would be available.

Lastly looking at the leader, the leader cannot serve it's own files as it is occupied with serving as the leader. Thus any exclusive files available at the DHT node become unreachable. It would be better if the Leader could transfer these chunks of files it has to other locations before taking the role of LEADER.