CS 179 - Senior Project in Operating Systems

Cabinet - Structured P2P using OpenDHT and FUSE

Eric Chan and Harini Venkatesan

Objective:

Our objective is to create a structured peer-to-peer filesystem using OpenDHT and FUSE. Traditional file systems are generally structured in a client-server format. For example, cloud services such as Google Drive are used to host and store files. Another approach is a peer-to-peer network, in which every participating user is both a client and server. Every user will be responsible for storing files, but every user also has the ability to distribute their file for storage. Some benefits of a peer-to-peer network are:

- Scalability: the more users the better the system performs.
- Decentralization: Users are free to join and leave the network with little repercussions. Additionally, there is no single point of failure and no overarching data authority.

Milestones:

- **OpenDHT (Independent unit)**

  Develop OpenDHT capabilities as an independent unit. Program communicates directly with OpenDHT. Capabilities include:
  - Communication between nodes (pings)
  - get() and put() operations for the hash table
  - Reaccessing a node (via private key and certificate)
    - QOL details, e.g. file location for storing private key and certificate

- **FUSE (Independent unit)**

  Develop FUSE capabilities as an independent unit. Program communicates directly with FUSE. Capabilities include:
  - Creating a filesystem
  - File posting (“upload”)
  - File retrieving (“download”)

- **Integrate OpenDHT and FUSE for File Retrieval**

  The goal is for the user to input a file request (by name) and receive the file seamlessly. See the corresponding *design details* section for details.

- **Integrate FUSE and OpenDHT for File Posting**

  The goal is for the user to transfer files to peers seamlessly. See the corresponding *design details* section for details.

- **Deploy using Docker (Final product)**

  Package final code into an abstracted container with all dependencies and requirements built in. Capabilities include:
  - Isolating the peer to peer file system to ensure it works uniformly
  - Standardizing docker image to be portable among different machines
  - Testing file sharing across multiple peers

Design Details:

The program has two primary components: (1) file posting and (2) file retrieval. We will first discuss file posting.

File Posting

When a user wishes to post a file, the program will ping peers in an unstructured way. When the program receives *x* pings back, say 3, then the program will send the file to those peers via FUSE. Upon receiving the file, the users will write/append to the hash table with *key:* file name and *data:* peer information. Thus, when the file is successfully transferred, the corresponding data entry should contain the information of 3 peers. The program will report back to the user *y* successes and *z* fails.

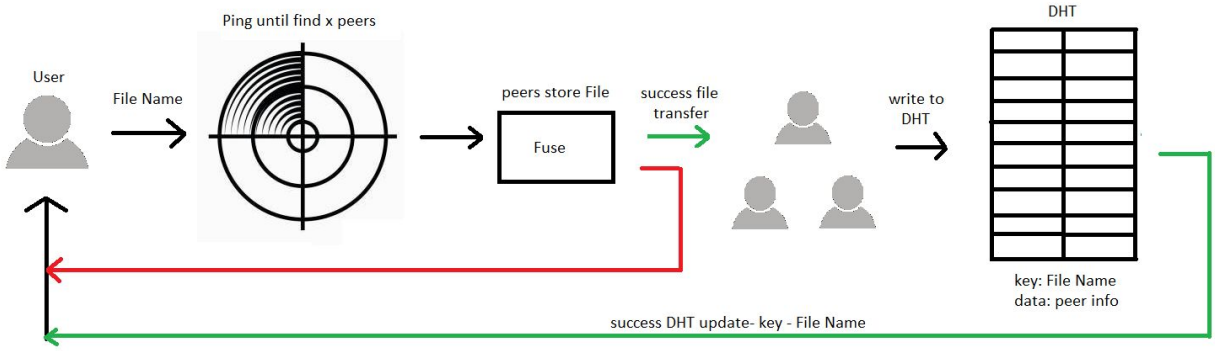

Figure 1. High level model for file posting

File Retrieval

When a user wishes to retrieve a file, the program will search the DHT with the file name as the key and fetch the corresponding data (peer information). The program then queries these peers for the file via FUSE. The user should receive their file. We use an auxiliary algorithm to determine if peers have left. This is described below.

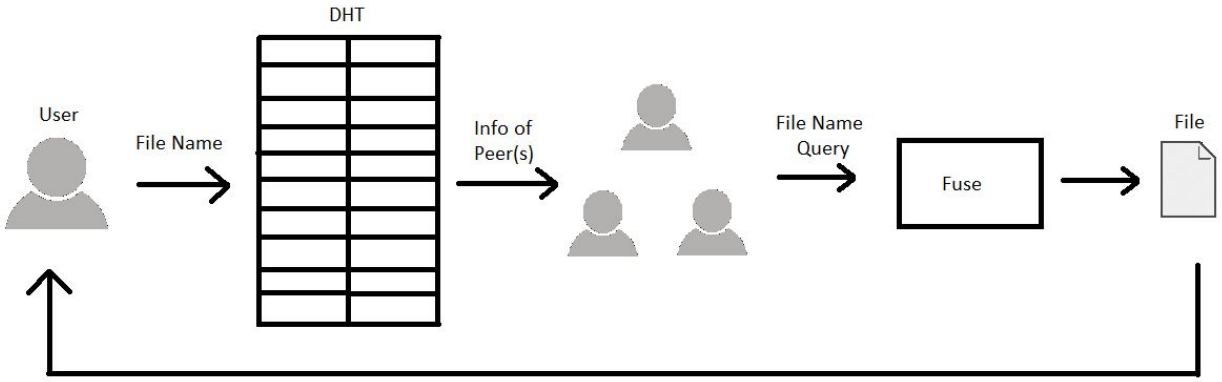

Figure 2. High level model for file retrieving

Auxiliary Algorithm

The purpose of this algorithm is to determine if a peer has left the network. This is only done upon file retrieval. Again, suppose the user's file is stored by 3 peers. The program runs a sub instance that intends to receive 3 files, but will only return 1 (the user only wants 1 copy).

If all 3 peers send the file, nothing happens. Suppose a peer does not send a file back. We want to add a strike to this peer as we suspect they may have left the network. To do this, we have keys in the DHT that correspond to each node. The corresponding data is the number of strikes for a node. Thus, if a neighbor does not return a file, then the program goes to add a strike to that node in the DHT.

If this node receives a certain capacity number of strikes, say 10, our program assumes they have left. What we do is delete this node from the DHT and call a subroutine to find another peer to store the original file (see File Posting).

For this auxiliary algorithm to work, there are other details we need to adjust. This can be how this sub instance runs, how to give a node a DHT entry, and what happens if a program instance does not see a node entry. However, these are all fairly easily to solve, so we will not describe these details here.

Github Link: https://github.com/harini-venkatesan/Cabinet.git