CSCI 5105 Introduction to Distributed Systems

Programming Assignment 1

(Simple File System on a DHT)

Arjun Subrahmanyan (Student ID : 5217513)
Devavrat Khanolkar (Student ID : 5211324)

Introduction

The following is a **design document** that describes the different components of the Simple File System on a DHT (based on **Chord** protocol) that was implemented in **Java** using **Apache Thrift**.

Building the DHT

To build the underlying system, the **Chord Join** protocol was implemented, through parts of the **SuperNode** and **Node** services. Following are the components of the service:

1. *join(IP, Port)*
   When the node wants to join the system, it invokes this function. The SuperNode has a flag ***busy***, which is set to **false** initially. When a node contacts the SuperNode to join the system, it sets this flag to **true**, implying no other node can join the system until the existing request has been completed. Any other node that contacts the SuperNode when it is busy receives a "**NACK**" message. If it receives a "**NACK**", then the thread sleeps for 1 second, before issuing a join call once again.

   The SuperNode then assigns an ID to the node by hashing it's IP address and port number (details of the hashing in the section Miscellaneous Features). The SuperNode maintains a list of assigned IDs and hence collision is resolved by selecting the next available ID in the hash space.

   Finally, the SuperNode updates the list of nodes in the system, through the string **nodeList** and returns it to the node that invoked *join()*.

2. *updateDHT(nodesList)*
   Once the SuperNode returns **nodesList**, this method is called on every node in the system. For nodes already existing in the system, this method just updates the respective finger table. The finger table of each node is maintained as a map between the i^{th} entry and the corresponding node. For the newly joined node, parameters like its IP, port and ID are set, before determining its predecessor and finger table. The corresponding methods within **NodeHandler** class that accomplish this are: *findPredecessor() and findSuccessor()*

3. *postJoin(IP, Port)*
   This method is invoked by the node, once *updateDHT()* is invoked on all the nodes. It resets the **busy** flag of the SuperNode to **false**.

Writing Files to the DHT

This component is implemented as part of the **SuperNode** and **Node** services through the following methods:

1. *getNode()*
   In this method, the SuperNode picks a random node from the existing nodes in the system and returns it to the client. This serves as the entry point for all read/write to the DHT by the client. Note that the entry point is fixed for all the requests (and is pre-determined even before the first request comes in).

2. *write(filename, contents)*
   This is the method used by the client to write a file in the DHT. The only parameters passed are the filename and the contents of the file. Since, we require tracking of the nodes visited during each write operation (and Thrift does not support the concept of method overloading), this method invokes an auxiliary method, *writeAux()*, which is described below.

3. *writeAux(filename, contents, list<> visitedNodes)*
   This is a recursive method that first determines the node responsible for the file, and then writes the contents into the file. The filename is first hashed to a **key** value and then a method *search()* is invoked, which returns the next node in the routing path. This is accomplished by *findNextNode()* which analyzes the finger table of the current node to determine the next hop. Each node that receives the request, appends its own ID into the list **visitedNodes**

   If *search()* returns nothing (**null)**, then the current node is where the file would be written. A folder is created locally that has its name in the format **folderX**, where **X** is the ID of the node. A file is created and the contents are written in it (file is overwritten if it already exists). An update is also made to the log file associated with the node (refer to section Miscellaneous Features).

   Else, the write request is forwarded to the non-null node returned by *search()*, i.e, *writeAux()* of the next node in the routing path is invoked.

Reading files from the DHT

This component is implemented as parts of the **SuperNode** and **Node** services through the following methods:

1. *getNode()*
   Same as the description in the previous section

2. *read(filename)*
   Similar to *write()*, this is a method that just invokes an auxiliary method *readAux()*, which also tracks the nodes visited to complete the read request.

3. *readAux(filename, list<> visitedNodes)*
   This is a recursive method that works similar to *writeAux()*. Again, the filename is hashed to a **key** value and then *search()* is invoked to find the next hop in the routing path. Also, the current node appends its own ID into **visitedNodes**.

   If the current node is the one responsible for the filename, then it attempts to open the file in the folder that it created while writing the same (or other) files. If no file is found, then we return a message **"File not found!"** back to the client. If the file could be opened and read successfully, then the contents are returned to the client as a string, which is displayed on the console. In this case, the request routing and tracking is logged.

   If the current node is not responsible for the filename, then it re-routes the request to the next hop in the path, i.e., it invokes *readAux()* of the next node, achieving the recursion.

Miscellaneous Features

Following section describes some additional features implemented as part of this assignment:

1. **The Structure of the DHT**: *getNodeDetails()*
   This is a method implemented as part of the **Node** service. Once a node has joined the system, the corresponding NodeClient can invoke this method (provided as an option in NodeClient.java), to get the details of the corresponding node. The details (printed on the console are): Node ID, Range of keys that the node is responsible for, the predecessor and successor, the file list and also its finger table.

   This method can be invoked on all the NodeClient processes to get the entire structure of the DHT.

2. **Logging the read/write requests** : *getLogs()*
   This is a method implemented as part of the **SuperNode** service. The Client can request for the logs of all the read and write operations requested. To do this, the client contacts the SuperNode (through *getLogs()*). The SuperNode, which has the list of all nodes in the system, then contacts each node for the respective logs (through the *getLogs()* method). The SuperNode then assembles all of this logged information and returns it back to the Client (displayed on the console).

   Each node maintains a log of all the write and read requests for files that it was responsible for. This includes a timestamp of the request, the filename and the nodes that were visited during the routing of the request from the entry point of the system.

3. **Hashing** : the *HashService* class
   This class has a static method *hash()*, which uses the MD5 algorithm to produce a hash value, in the range of [**0, Maximum number of nodes in the system**). This method is used in two separate components of the program:

   - When a node wants to join the system, it is assigned an ID by the SuperNode. This assignment is done by hashing the IP and Port number of the node
   - When the Client wants to read a file from the DHT or write a file into the DHT, then the filename is hashed to the same space, in order to identify the node responsible for it.

4. **Deleting the file system** : exit() in SuperNode and cleanup() in Node
   When the client exits by inputting a 0, the exit() method in the SuperNode is called, which calls the cleanup() method on all nodes currently connected. The cleanup() method then deletes all the files which have been written by the client to that node.

Other possibilities

The nodes and filenames are hashed to an m-bit identifier using the hash function, as described above. This parameter m decides the total number of nodes in the system, which is namely 2^m. In this implementation, we have included an enum data type, **Constants**, that bears a constant named **MAX_NODES** and is set the value of 16. This implies that $m = 4$ and there will be 4 entries in the finger table of each node. This parameter can instead be provided in a configuration file, which the system can read when it starts.