Project 3: Distributed Hash Table - Join and Remove

Due: November 6th, 2014 23:59:59pm

In this project, you will add join and remove functionality to the basic distributed hash table (DHT) [1] that you created in Project 2. As in Project 2, the DHT will take the form of an Apache Thrift Service. Put and get functions in DHT will be implemented as the writeFile and readFile functions.

I have provided a file, `chord_join_remove_dht.thrift`, that defines the operations needed for this project.

The project consists of three parts, described below. In addition, the `join` and `remove` functions must be executed by sending remote procedure calls to running server instances. The code you use to make these RPCs does not need to be submitted. But you will need to write the code to debug your program.

1 Compile the interface definition file

As in Project 2, I have provided an Apache Thrift interface definition file, `chord_join_remove_dht.thrift`. It can be downloaded from Blackboard. You can use Thrift to compile interface definition file to either Java, C++, or Python.

2 Extend the server-side method stubs generated by Thrift

You must implement functions corresponding to a number of server-side methods:

The following 6 server-side functions should be implemented the same as in Project 2

writeFile given a name, owner, and contents, the corresponding file should be written to the server. Meta-information, such as the owner, creation-time, update-time, version, content length, and content hash (use the **SHA-256 hash**) should also be stored at the server side. If the filename does not exist on the server, a new file should be created with its version attribute set to 0. Otherwise, the file contents should be overwritten and the version number should be incremented.

readFile if a file with a given name and owner exists on the server, both the contents and meta-information should be returned. Otherwise, a SystemException should be thrown and appropriate information indicating the cause of the exception should be included in the SystemException's message field.

deleteFile if a file with a given name and owner exists on the server, it should be deleted, but the metadata should be retained and the deleted timestamp modified to record the time of deletion. If an error occurs during file deletion, e.g., deleting a non-existent file, a SystemException should be thrown.

findSucc given an identifier in the DHT's key space, returns the DHT node that owns the id. This function should be implemented in two parts:

1. the function should call `findPred` to discover the DHT node that precedes the given id
2. the function should call `getNodeSucc` to find the successor of this predecessor node

findPred given an identifier in the DHT's key space, this function should returns the DHT node that immediately precedes the id. This preceeding node is the first node in the counter-clockwise direction in the Chord key space. A SystemException should be thrown if no fingertable exists for the current node.

getNodeSucc returns the closest DHT node that follows the current node in the Chord key space. A SystemException should be thrown if no fingertable exists for the current node.

The following 7 additional functions should be used to add and remove nodes from an existing DHT

pullUnownedFiles returns a list of files that are held by a node but no longer owned by the node. These files should be deleted before the function returns (do not worry about consistency).

pushUnownedFiles takes a list of `rFiles` that are held by the node calling this function but no longer owned by the calling node. The node that this function is invoked on should store the files passed in the argument.

updateFinger given a fingertable entry index, i, and a NodeID, $node$, this function updates the i^{th} fingertable entry to $node$.

getFingertable this function should return the fingertable (a list of nodes). This function is included to aid in debugging. In addition, the automated grading scripts will use this function to compare fingertables updated incrementally, with fingertables created using the `cmp_fingertables` code from Part 3.

setNodePred sets the current node's predecessor to the $node$ provided in the argument of the function.

join given a node in an existing DHT, this method should add the server that receives the RPC, referred to as $node_{new}$, to the DHT.

The join process involves 3 steps (See Figure 6 in the Chord paper [1]):

1. Create a fingertable for the new node.
2. Update the fingertables of the nodes affected by the addition of $node_{new}$. To perform these updates, search for $p = pred(node_{new} - 2^i)$ (where i is an index in the fingertable). If $p + 2^i \in (pred(node_{new}), node_{new}]$, then the i^{th} fingertable entry of p should be updated to $node_{new}$. To update the fingertable of p, the RPC `updateFinger` should be called from $node_{new}$.
3. Transfer the files that are owned by $node_{new}$ but held by the successor to $node_{new}$. This tranfer portion should use `setNodePred` to update the predecessor of the $node_{new}$'s successor to $node_{new}$ then use `pullUnownedFiles` to take ownership of the files that it now owns.

If no nodes exist in the DHT, then the `join` function should be called with a null argument. In this case, only $node_{new}$'s fingertable needs to be created, and it all of these fingertable entries should contain $node_{new}$.

remove removes the server that receives the RPC, referred to as $node_{remove}$, from the DHT. The remove process involves 2 steps:

1. Update the fingertables of the nodes affected by the removal of $node_{remove}$. This process should be similar to step 2 in the `join` function. These fingertable updates should call the function `updateFinger`.
2. Use `pushUnownedFiles` to transfer the files owned by $node_{remove}$ to its successor.

The remove method should throw SystemException if it is the only node in the DHT.

To properly implement `join` and `remove`, I recommend reading the reference materials (Chapter 5.2.3 in the textbook and the Chord paper [1]). The Chord paper describes an $O(\log^2(n))$ implementation of these methods, which is okay for this project.

According to the Chord protocol, each Chord node is responsible for a portion of the Chord key space **(see [1])**. As in Project 2, a node's id is determined by the SHA-256 hash value of the string "<public ip address>:<port number>". A file's id is determined by the SHA-256 hash value of the string "<owner>:<filename>". **Attempts to** `write`**,** `read`**, or** `delete` **a file from a node that is not the successor of the file's associated SHA-256 id should throw a *SystemException*.**

Note: When accessing `remote.cs.binghamton.edu`, you are actually redirected to one of the 16 `REMOTE` machines ({remote00, remote01, ..., remote09, remote10, ..., remote15}.cs.binghamton.edu) using DNS redirection. So you need to make sure you use the public IP address of the remote machine that the server is running on. For example, the public IP of `remote01.cs.binghamton.edu` is `128.226.180.163`.

As in Projects 1 and 2, every time the server is started, any filesystem or other persistent storage that it used should be initialized to be empty. If your server stores files in the file system, it should use only the current working directory from where it was run. There is no need to implement directory structure.

In addition, the server executable should take a single command-line argument specifying the port where the Thrift service will listen for remote clients. For example:

```
./server 9090
```

It should use Thrift's TBinaryProtocol for marshalling and unmarshalling data structures. Like Project 2, multiple servers will be running at the same time. **These servers should use Thrift's** `TSimpleServer`**. Because** `TSimpleServer` **can handle at most one connection at a time, you must take care to prevent a node from accidentally invoking an RPC on itself, causing a deadlock.**

3 Debug your code with the compare fingertables program

To assist in debugging your code, I have provided a compare fingertables program, `cmp_fingertables`, which will print the correct fingertables of a list of nodes. Similar to the `init` program I provided in Project 2, the `cmp_fingertables` program takes a filename as its only command line argument. To run the compare fingertable program, type the following commands:

```
$> chmod +x cmp_fingertables
$> ./cmp_fingertables nodes.txt
```

The file (`nodes.txt`) should contain a list of IP addresses and ports, in the format "<public-ip-address>:<port>", of all of the running DHT nodes.

For example, if four DHT nodes are running on `remote01.cs.binghamton.edu` port 9090, 9091, 9092, and 9093, then `nodes.txt` should contain:

```
128.226.180.163:9090
128.226.180.163:9091
128.226.180.163:9092
128.226.180.163:9093
```

4 How to submit

To submit the project, you should first create a directory whose name is "your BU email ID"`-project3`. For example, if your email ID is `jdoe@binghamton.edu`, you should create a directory called `jdoe-project3`.

1. The `chord_join_remove_dht.thrift` file I provided.

2. Your source code, which should include at least one file implementing the server. (You no longer need to submit your code implementing the client.)

3. A `Makefile` to compile your source code into one executable, which should be named `server`. (It is okay if this executable is a bash script that calls the Java interpreter, as long as the command line arguments follow the format described in Part 2.)

4. A `Readme` file describing the programming language(s)/tools (e.g., Apache Ant) that you used and the implementation details of your program.

Compress the directory (e.g., `tar czvf jdoe-project3.tgz jdoe-project3`) and submit the tarball (in `tar.gz` or `tgz` format) to Blackboard. Again, if your email ID is `jdoe@binghamton.edu`, you should name your submission: `jdoe-project3.tar.gz` or `jdoe-project3.tgz`. Failure to use the right naming scheme will result in a 5% point deduction.

If you used Eclipse, please delete the project's `.metadata` directory before creating the tarball.

Your project will be graded on the CS Department computers `remote.cs.binghamton.edu`. It is your responsibility to make sure that your code compiles and runs correctly on CS Department computers. If you use external libraries in your code, you should also include them in your directory and correctly set the environment variables using absolute path in the Makefile. If your code does not compile on or cannot correctly run on the CS computers, you will receive no points.

References

[1] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, August 2001.