Project 4: Stable Distributed Hash Table

Due: December 2nd, 2014 23:59:59pm

In this project, you will implement remote procedure calls to allow stable Distributed Hash Table (DHT) operation when nodes join the DHT concurrently.

As in the previous projects, the DHT will take the form of an Apache Thrift Service. Unlike the previous projects, you must also write code to verify that your stable DHT operates correctly.

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

The project consists of two parts, described below.

1 Compile the interface definition file, and extend the server-side method stubs generated by Thrift

As in the previous projects, I have provided an Apache Thrift interface definition file, `stable_dht.thrift`, downloadable from Blackboard. You can use Thrift to compile interface definition file to either Java, C++, or Python.

Unlike previous projects, you are no longer required to implement file server methods.

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

The following 3 server-side functions should be implemented the same as in previous projects.

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

getNodeSucc returns the closest DHT node that follows the current node in the Chord key space.

getFingertable this function should return the fingertable (a list of nodes). This function will be required for the test portion of the project.

The implementation of following 1 server-side function should be changed from Project 3.

findPred given an identifier in the DHT's key space, this function should return the DHT node that immediately precedes the id. This preceding node is the first node in the counter-clockwise direction in the Chord key space. **If the associated fingertable entry has not been set, the function should attempt to contact the preceding entry in the fingertable until a valid entry is found.** The result of the findPred operation may be incorrect if the fingertable entries have not yet been set by the stable Chord protocol. You are not required to detect this case for this project.

The following 5 additional functions should be implemented as described in the Chord paper [1] (Figure 7), to ensure that all nodes eventually discover their correct successors.

getNodePred the function should return the current node's predecessor. This predecessor should be stored by the node and initialized to `NULL` (or a structure indicating the predecessor has not been set). This stored predecessor is not guaranteed to be correct but is updated by `stabilize` and `notify` so that all predecessors and successors are eventually set to their correct values.

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. Unlike Project 3, this method does not attempt to fully create a fingertable for the new DHT node (fingertable entries are continuously updated by the `fixFingers` function). It simply finds and stores the new node's successor.

stabilize this function is run at intervals in each DHT node (it should be called from within a separate thread than the main server thread). `stabilize` should check if a DHT node exists between the current node, n, and its current successor, $n.successor$. It does so by checking if $n.successor.predecessor \in (n, n.successor)$. If such an intermediate node exists, then the node calls `notify` on the intermediate node to set its predecessor to the current node.

If the predecessor of the node n's successor has not been set, then `stabilize` should `notify` $n.successor$ that n is the predecessor. **(Note that the case where the predecessor has not been set is not covered in the Chord paper, Figure 7.)**

notify given a node, n', this function sets the predecessor of the current node, n, to n' if $n' \in (n.predecessor, n)$.

fixFingers this function is run at intervals in each DHT node (it can be run from the same thread as `stabilize`). This function chooses a fingertable entry uniformly at random and updates it to the successor of the entry's id.

If you have not done so already, read the Chord paper [1]) to gain an understanding of the level of consistency that the stabilized version of the Chord DHT provides.

According to the Chord protocol, each Chord node is responsible for a portion of the Chord key space **(see [1])**. As in earlier projects, a node's id is determined by the SHA-256 hash value of the string "<public ip address>:<port number>".

Note: When accessing `remote.cs.binghamton.edu`, you are 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`.

The server executable should take **two** command-line arguments. The first should specify the port where the Thrift service will listen for remote clients and the second should specify r, the time interval (in seconds) between calls to the `stabilize` function. For example:

```
./server 9090 1
```

would start the server on port 9090 with a one second interval between calls to stabilize at each node. The server program should use Thrift's TBinaryProtocol for marshalling and unmarshalling data structures.

As in Projects 2 and 3, multiple servers will be running at the same time. However, multiple calls to `findPred` are expected to run concurrently on the network. These concurrent calls could cause a deadlock if `TSimpleServer` is used, so the DHT nodes for this project should use either `TThreadedServer` or `TThreadPoolServer`.

2 Check the performance of your stabilized DHT implementation

Implement a `test` program to check that after a sufficient amount of time, all successors in a Chord DHT are correct.

The program should take three arguments:

1. $port$ should be the starting port number for the spawned DHT nodes
2. r should be the interval between calls to `stabilize`
3. N should be the number of nodes to spawn

For example, when the following command is issued

```
./test 9090 1 5
```

The `test` program should spawn 5 DHT nodes listening on ports 9090, 9091, 9092, 9093, and 9094 with a 1 second interval between calls to `stabilize` for each node. **The `test` program can run these nodes by calling methods within your server code. It does not need to spawn separate server processes.**

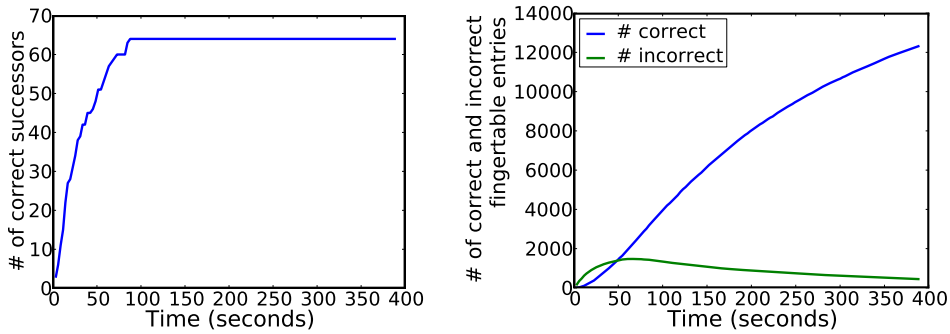
After spawning the specified number of nodes, the test program should call the `getFingertable` and `getPredNode` functions on all N DHT nodes.

The test program should then print out the following information:

1. the number of correct predecessors
2. the number of correct successors (the first entry in the fingertable)
3. the total number of correct fingertable entries (the true fingertable can be calculated by calling `cmp_fingertables` provided in Project 3)
4. the total number of incorrect fingertable entries that have been set by `fixFingertable` (don't include fingertable entries that have not been set as incorrect entries.)

The test program should repeat the `getFingertable` and `getPredNode` calls and print the associated output at intervals less than r seconds (the exact time between queries is not important) until all fingertable entries are correct.

If you correctly implemented the Stable Chord protocol, then the number of correct successors and fingertable entries should increase while the number of incorrect fingertable entries should first rise and then fall as the DHT stabilizes. Plots of these trends are shown in Figure 1.



(a) (b)

Figure 1: **(a)** shows the number of correct successors over time in a DHT of 64 nodes. **(b)** shows the number of correct and incorrect fingertable entries over time in a DHT of 64 nodes.

2.1 EXTRA CREDIT (optional, 30%)

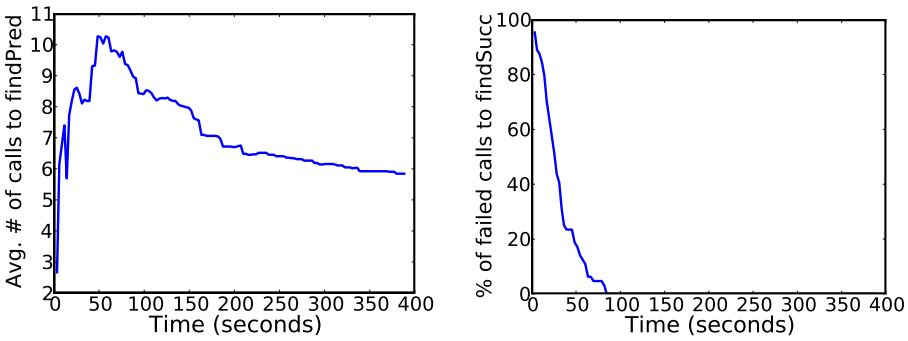
For optional extra credit, add an experiment to your `test` program to gauge the performance of the Stable Chord protocol.

For this experiment, in addition to calling `getFingertable` and `getNodePred`, your `test` program should also issue N calls to `findSucc`. Specifically, `findSucc` should be called on node n to retrieve the successor of key $n.id$ for all nodes in the DHT. This should result in the maximum number of recursive calls through `findPred` (`findSucc` **should be implemented as in the previous projects, to first issue a call the** `findPred` **then return the successor of the returned predecessor.**).

The `test` program should then output the following information:

1. the average number of correct successor nodes returned by `findSucc`.
2. the average number of calls to `findPred` for each correct successor returned (Use the optional `count` field in the `NodeID` structure to compute the number of `findPred` calls were needed.).

If you correctly implemented the Stable Chord protocol, then the number of incorrectly returned successors should decrease as the DHT stabilizes. In addition, the average number of call to `findPred` should first rise as more nodes become discoverable in the DHT then fall as fingertable entries become correct. Plots of these trends are shown in Figure 2.



(a) (b)

Figure 2: **(a)** shows the average number of calls to `findPred` needed to recover a correct successor in a DHT of 64 nodes. **(b)** shows the percentage calls to `findSucc` over time that return an incorrect successor in a DHT of 64 nodes.

3 How to submit

To submit the project, you should first create a directory whose name is "your BU email ID"`-project4`. For example, if your email ID is `jdoe@binghamton.edu`, you should create a directory called `jdoe-project4`.

You should put the following files into this directory:

1. The `stable_chord.thrift` file I provided.
2. The `gen-cpp`, `gen-java`, or `gen-py` directory generated from the thrift IDL file.
3. Your source code, which should include at least one file implementing the `server` and one implementing the `test` program.

4. A `Makefile` to compile your source code into two executables. The which should be named `server` and `test`. (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.)

5. A `Readme` file describing the programming language(s)/tools (e.g. Apache Ant) that you used, the implementation details of your program, sample input/output of your program, and **whether you have elected to complete the extra credit portion**.

Compress the directory (e.g., `tar czvf jdoe-project4.tgz jdoe-project4`) 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-project4.tar.gz` or `jdoe-project4.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. You should set the environment variables such as PATH using absolute path in your Makefile (e.g., cpp or java), your code (e.g., python), or build.xml (e.g., java ant). 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 your Makefile (e.g., cpp or java), your code (e.g., python), or build.xml (e.g., java ant). 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.