CMSC 417 Computer Networks **Fall 2019**

Programming Assignment 4

Assigned: October 17 *Due: November 7, 11:59:59 PM.* *Weight: 1.75x*

1 Introduction

In this project, you will implement Chord [1]. To do so, you must read the paper describing the Chord protocol, algorithms, and implementation, which is available at the following URL:

`http://www.cs.berkeley.edu/~istoica/papers/2003/chord-ton.pdf`

Chord is a peer-to-peer lookup protocol which enables the mapping of a key (or identifier) to a peer (or node) in the network. Chord exposes a single function, "lookup", to higher-layer applications:

lookup(<Key>) → <Host>

Chord uses local information and communicates with other peers to find the host (IP address and port) that a given key is mapped to.

Applications may then be built on top of this service that Chord provides. For example, a distributed hash table (DHT) application may distribute the key-value storage of a hash table across many peers and support the typical operations (i.e., insert, get, and remove). The DHT may be implemented on top of Chord by storing the key-value pairs at the node that Chord mapped to the given key.

2 Protocol

The Chord protocol and algorithms are described in the paper [1]. **You should read the paper to learn about the design of Chord, prior to starting your own implementation.** The implementation in the paper, shown in Figure 5 and Figure 6, is presented as pseudocode. The pseudocode involves both local and remote function call invocations, determined by the node at which a function call is invoked. Note that this pseudocode is extended by Section IV.E.3 "Failure and Replication", with changes to the 'stabilize', 'closest_preceding_node', and 'find_successor' function calls. Additionally, there is a new function call 'get_successor_list' hinted at in the description, which returns a node's list of successors and is used in the extended function calls.

For this project, you may safely ignore the following portions of the paper:

- Section IV.E.4 "Voluntary Node Departures"
  (All node departures will be treated as node failures.)
- Section II "Related Work"
- Section V "Evaluation"
- Section VI "Future Work"

- Section VII "Conclusion"

As discussed in the paper, there are two ways to implement the protocol: iteratively or recursively. Figure 5 in the paper presents the "find_successor" function using a recursive implementation. However, it may be easier to approach it iteratively, since then each node will be able to respond to any incoming RPCs immediately without blocking to wait on responses from other nodes. The following website presents an iterative implementation of the pseudocode for Figure 5 in the paper that you may find helpful (see "Iterative lookups"):

`https://cit.dixie.edu/cs/3410/asst_chord.html`

3 Chord Client

The Chord client will be a command-line utility which takes the following arguments:

1. `-a <String>` = The IP address that the Chord client will bind to, as well as advertise to other nodes. Represented as an ASCII string (e.g., 128.8.126.63). Must be specified.
2. `-p <Number>` = The port that the Chord client will bind to and listen on. Represented as a base-10 integer. Must be specified.
3. `--ja <String>` = The IP address of the machine running a Chord node. The Chord client will join this node's ring. Represented as an ASCII string (e.g., 128.8.126.63). Must be specified if `--jp` is specified.
4. `--jp <Number>` = The port that an existing Chord node is bound to and listening on. The Chord client will join this node's ring. Represented as a base-10 integer. Must be specified if `--ja` is specified.
5. `--ts <Number>` = The time in milliseconds between invocations of 'stabilize'. Represented as a base-10 integer. Must be specified, with a value in the range of [1,60000].
6. `--tff <Number>` = The time in milliseconds between invocations of 'fix_fingers'. Represented as a base-10 integer. Must be specified, with a value in the range of [1,60000].
7. `--tcp <Number>` = The time in milliseconds between invocations of 'check_predecessor'. Represented as a base-10 integer. Must be specified, with a value in the range of [1,60000].
8. `-r <Number>` = The number of successors maintained by the Chord client. Represented as a base-10 integer. Must be specified, with a value in the range of [1,32].
9. `-i <String>` = The identifier (ID) assigned to the Chord client which will override the ID computed by the SHA1 sum of the client's IP address and port number. Represented as a string of 40 characters matching [0-9a-fA-F]. Optional parameter.

An example usage to start a new Chord ring is:

```
chord -a 128.8.126.63 -p 4170 --ts 30000 --tff 10000 --tcp 30000 -r 4
```

An example usage to join an existing Chord ring is:

```
chord -a 128.8.126.63 -p 4171 --ja 128.8.126.63 --jp 4170 --ts 30000 --tff 10000 --tcp 30000 -r 4
```

The Chord client will open a TCP socket and listen for incoming connections on port specified by `-p`. If neither `--ja` nor `--jp` is specified, then the Chord client starts a new ring by invoking 'create'. The Chord client will initialize the successor list and finger table appropriately (i.e., all will point to the client itself).

Otherwise, the Chord client joins an existing ring by connecting to the Chord client specified by `--ja` and `--jp` and invoking 'join'. The initial steps the Chord client takes when joining the network are described in detail in Section IV.E.1 "Node Joins and Stabilization" of the Chord paper.

Periodically, the Chord client will invoke various stabilization routines in order to handle nodes joining and leaving the network. The Chord client will invoke 'stabilize', 'fix_fingers', and 'check_predecessor' every `--ts`, `--tff`, and `--tcp` milliseconds, respectively.

Commands The Chord client will handle commands by reading from `stdin` and writing to `stdout`. There are two command that the Chord client must support: 'Lookup' and 'PrintState'.

'Lookup' takes as input an ASCII string (e.g., "Hello"). The Chord client takes this string, hashes it to a key in the identifier space, and performs a search for the node that is the successor to the key (i.e., the owner of the key). The Chord client then outputs that node's identifier, IP address, and port.

'PrintState' requires no input. The Chord client outputs its local state information at the current time, which consists of:

1. The Chord client's own node information
2. The node information for all nodes in the successor list
3. The node information for all nodes in the finger table

where "node information" corresponds to the identifier, IP address, and port for a given node.

An example sequence of command inputs and outputs at a Chord client is shown below. There are four participants in the ring (including the Chord client itself) and -r is set to 2. You may assume the same formatting for the input, and must match the same formatting for your ouput. ">" and "<" represent `stdin` and `stdout` respectively. The input commands will be terminated by newlines ('\n').

```
> Lookup Hello
< Hello f7ff9e8b7bb2e09b70935a5d785e0cc5d9d0abf0
< 31836aeaab22dc49555a97edb4c753881432e01d 128.8.126.63 2001

> Lookup World
< World 70c07ec18ef89c5309bbb0937f3a6342411e1fdd
< 7d157d7c000ae27db146575c08ce30df893d3a64 128.8.126.63 2003

> PrintState
< Self 31836aeaab22dc49555a97edb4c753881432e01d 128.8.126.63 2001
< Successor[1] 6fa8c57336628a7d733f684dc9404fbd09020543 128.8.126.63 2002
< Successor[2] 7d157d7c000ae27db146575c08ce30df893d3a64 128.8.126.63 2003
< Finger[1] 6fa8c57336628a7d733f684dc9404fbd09020543 128.8.126.63 2002
< Finger[2] 6fa8c57336628a7d733f684dc9404fbd09020543 128.8.126.63 2002
...
```

```
< Finger[159] 7d157d7c000ae27db146575c08ce30df893d3a64 128.8.126.63 2003
< Finger[160] f4d60480373006cb24147cd17765000f14aadca3 128.8.126.63 2004
```

4 Implementation

We are providing you with some infrastructure to help you get started on this project. You may find the supporting files in the 'assignment4' directory in the 'materials' repository. Please rememeber to run 'git pull' to fetch the latest version of the materials.

In this project, we will be using Google Protocol Buffers (protobuf) for handling the messages passed between nodes. We are including the full specification for the Chord protocol, which you can find in the 'chord.proto' file. The 'Call' message type specifices the 'name' of the function to execute along with a byte array 'args' containing the serialized/packed message for the '*Args' message type that corresponds to the function. The supported function names are as follows: "find_successor", "notify", "get_predecessor", "check_predecessor", and "get_successor_list". The 'Return' message type specifies whether the function was executed successfully or not ('success') and, if true, may include an optional byte array 'value' that contains the serialized/packed message for the return value from the function (according to the '*Ret' message type).

For an example of how to use protobufs in C, please refer to the example 'rpc.proto' and 'rpc.c' files which demonstrate an example of handling a function 'bool invert(bool)' via protobufs. This example uses the same Call/Return message framing as the full Chord specification, so you will be able to reuse a significant portion of this code. The included 'Makefile' will build a 'rpc' executable, which invokes the 'invert' function for inputs 0 and 1.

Note that this example simply showcases local handling of the RPC; you are responsible for exchanging the serialized protobufs with remote peers over the network. To do this, we expect you to use a simple packet structure that contains a 64-bit integer (in network byte order) that is set to 8 plus the length of the serialized bytes, followed by the payload consisting of the serialized bytes themselves.

The directory also contains OpenSSL wrapper functions in 'hash.c' and 'hash.h' for computing a SHA1 hash, with usage demonstrated in 'hashtest.c'. The included 'Makefile' will build a 'hashtest' executable, which invokes the SHA1 hashing functions to compute the hashes for "Hello" and "World". Note that the output should match the example in the previous section.

The identifier for a node should be the SHA1 sum of the node's IP address and port number, but how the IP address and port number are input to the hash function is up to you (e.g., as a string, or byte array).

5 Grading

Your project grade will depend on the parts of the project that you implement. Assuming each part has a "good" implementation, the grades are as follows:

Grade	Parts Completed
A	Resilient to < (-r) node failures.
B	Uses finger tables to route efficiently.
C	Handles many nodes joining with updates to successor lists. Makes routing choices based on successor lists.

We will only test your client against other instances of your client.

6 Additional Requirements

1. Your code must be submitted as a series of commits that are pushed to the origin/master branch of your Git repository. We consider your latest commit prior to the due date/time to represent your submission.

2. The directory for your project must be called 'assignment4' and be located at the root of your Git repository.

3. You must provide a Makefile that is included along with the code that you commit. We will run 'make' inside the 'assignment4' directory, which must produce a 'chord' executable also located in the 'assignment4' directory.

4. You must submit code that compiles in the provided VM, otherwise your assignment will not be graded.

5. Your code must be -Wall clean on gcc/g++ in the provided VM, otherwise your assignment will not be graded. Do not ask the TA for help on (or post to the forum) code that is not -Wall clean, unless getting rid of the warning is the actual problem.

6. You are not allowed to work in teams or to copy code from any source.

References

[1] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications. *Networking, IEEE/ACM Transactions on*, 2003.