Project 1: Chord

Due: 11:59 PM, Feb 14, 2017

Contents

1 Introduction

Welcome to CS 138! This project is a self-contained introduction to many of the concepts that you'll be using frequently in CS 138. You'll be implementing a real distributed system while getting some practice using RPCs and concurrency control mechanisms in Go. Before you dive into this project, make sure you've signed and turned in the collaboration policy and that you've read the coding guidelines that we've posted on the website.

2 Chord

Chord is one of the first distributed hash table (DHT) protocols proposed and developed by researchers at MIT in 2001 in the paper titled "Scalable Peer-to-peer Lookup Service for Internet

Applications"[1]. From an application's perspective, Chord simply provides a service that can store key-value pairs and find the value associated with a key reasonably quickly. Behind this simple interface, Chord distributes objects over a dynamic network of nodes, and implements a protocol for finding these objects once they have been placed in the network. Every node in this network is a server capable of looking up keys for client applications, but also participates as a key store. Hence, Chord is a decentralized system in which no particular node is necessarily a performance bottleneck or a single point of failure (if the system is implemented to be fault-tolerant).

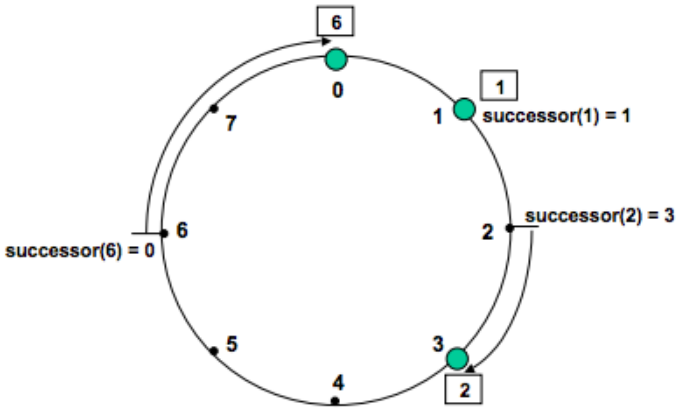
2.1 Keys

Every key inserted into the DHT must be hashed to fit into the key-space supported by the particular implementation of Chord. The hashed value of the key will take the form of an m-bit unsigned integer. Thus, the keyspace (the range of possible hashes) for the DHT resides between 0 and $2^m - 1$, inclusive. Standard practice for most DHT implementations is to use a consistent hash function, such as SHA-1 (or others), to produce a message digest of the size of m (e.g., 128 bits). Using these hashing algorithms ensures with high probability that the hashes generated from keys are distributed evenly throughout the keyspace. Note that this does not restrict the number of distinct keys that may be stored by the DHT, as the hash only provides a guide for locating the key in the network, rather than providing the identifier for the key. It is possible, though unlikely, for the hash values of distinct keys to collide.

2.2 The Ring

Just as each key that is inserted into the DHT has a hash value, each node in the system also has a hash value in the keyspace of the DHT. To get this hash value, we can simply give each node a distinct name and then take the hash of the name (e.g. IP address and port), using the same hashing algorithm we use to hash keys inserted into the DHT. Once each node has a hash value, we are able to give the nodes an ordering based on those hashes. Chord orders the nodes in a circular fashion, in which each node's successor is the node with the next highest hash. The node with the largest hash, however, has the node with the smallest hash as its successor. It is easy to imagine the nodes placed in a ring, where each node's successor is the node after it when following a clockwise rotation. This can be seen below, where the circle has three nodes: 0, 1, and 3. The successor of identifier 1 is node 1, so key 1 would be located at node 1. Similarly, key 2 would be located at node 3, and key 6 at node 0.

[1] `http://pdos.csail.mit.edu/papers/chord:sigcomm01/chord_sigcomm.pdf`



To locate the node at which a particular key-value pair is stored, one need only find the successor to the hash value of the key.

2.3 The Overlay Network

As the paper in which Chord was introduced states, it is possible to look up any particular key by sending an iterative request around the ring. Each node would determine whether its successor is the owner of the key, and if so perform the request at its successor. Otherwise, the node asks its successor to find the successor of the key and the same process is repeated. Unfortunately, this method of finding successors would be incredibly slow on a large network of nodes. For this reason, Chord employs a clever overlay network that, when the topology of the network is stable, routes requests to the successor of a particular key in $O(\log n)$ time, where n is the number of nodes in the network.

This optimized search for successors is made possible by maintaining a "finger" table at each node. The number of entries in the finger table is equal to m, where m is the number of bits representing a hash in the keyspace of the DHT (e.g., 128). Entry i in the table, with $0 \leq i < m$, is the node which the owner of the table believes is the successor for the hash $h + 2^i$ (h is the current node's hash). When node A services a request to find the successor of the key k, it first determines whether its own successor is the owner of k (the successor is simply entry 0 in the finger table). If it is, then A returns its successor in response to the request. Otherwise, node A finds node B in its finger table such that B has the largest hash smaller than the hash of k, and forwards the request to B.

2.4 Dynamic Addition & Removal

Chord would be far less useful if it were not designed to support the dynamic addition and removal of nodes from the network, requiring a static allocation of nodes instead. A production ready implementation of Chord would support the ability to add and remove nodes from the network at arbitrary times, as well as cope with the failure of some nodes, all without interrupting the ongoing client requests being served by the DHT. This functionality complicates the implementation considerably, though.

To allow membership in the ring to change, protocols for creating a ring, adding a node to the

ring, and leaving the ring must be defined. Creating the ring is easy. The first node fills its finger table with references to itself, and has no predecessor. Then, when node A joins the network, it asks an existing node in the ring to find the successor of the hash of A. The node returned from that request becomes the successor of A. The predecessor of A is undefined until some other node notifies A that it believes that A is its successor. In order to determine the successor and predecessor relationships between nodes as they are added to the network (and voluntarily removed), each Chord node performs `stabilize` and `fixNextFinger` periodically:

```
n - this node
h - hash of n
m - the number of bits in a hash

stabilize()
    x = successor.predecessor
    if x is between n and successor
        successor = x
    successor.notify(n)

fixNextFinger()
    // next is initialized to 1 external of this function
    next_hash = h + 2^(next) mod 2^m
    finger[next] = findSuccessor(next_hash)
    next = next + 1
    if next > m - 1
        next = 1

notify(p)
    if predecessor is null OR p is between predecessor and n
        predecessor = p
        transfer appropriate keys to predecessor
```

Note: For $fixNextFinger$, initializing next to 0 can also work, depending on your implementation.

2.5 Extra Credit: Fault Tolerance

For the standard assignment, we do not expect for you to have a fault tolerant solution. However, we do expect you to implement graceful node exits. This means that a node can exit the ring but can inform its predecessor and successor that it is doing so.

As extra credit, fault tolerance is achieved by maintaining successor lists, rather than a single successor so that the failure of a few nodes is not enough to send the system into disrepair. Keys must also be replicated across a number of nodes so that they are available in the event that some of the nodes storing them fail. The implementation details of replication, and the other problems that arise from this are beyond the scope of this assignment and will be touched on in later lectures/assignments. Since your keys are immutable you don't need to worry about updating existing replicas and any sort of invalidation, or issues related to atomic updates across multiple nodes.

In order to understand how the system works as a whole, it is important to see that although optimizations based on the finger table may not always be available, it is always possible to find the correct successor node for a given hash. This is an invariant of the system, even when nodes are joining and leaving the network with great frequency.

3 The Assignment

You will be implementing a simple version of Chord. Your implementation need not be fault tolerant (but should support graceful exits), and we are stressing correctness over performance in this assignment. To remove unnecessary complication, you should treat key-value pairs in the DHT as **immutable**. This means that once a key-value pair is inserted into the DHT, it cannot be deleted and the value associated with the key may not change (you should enforce this requirement). Your implementation of Chord should support dynamic insertion and removal of nodes, and continue to serve `get` and `put` requests simultaneously and correctly (if a value exists for a key, it must always be accessible). Keys should never reside at more than two nodes at any given time, and only one node when the ring is in a stable state.

Your TAs have written a significant amount of support code for you. The code you must write is marked with `//TODO students` comments and is spread across all the Go files in the `chord` directory, except for `chord/node_rpc_api.go`. Feel free to add helper functions but you must implement the following functions:

- `node.go`
  - `func (node *Node) join(other *RemoteNode) error`
  - `func (node *Node) stabilize(ticker *time.Ticker)`
  - `func (node *Node) notify(remoteNode *RemoteNode)`
  - `func (node *Node) findSuccessor(id []byte) (*RemoteNode, error)`
  - `func (node *Node) findPredecessor(id []byte) (*RemoteNode, error)`
- `chord.go`
  - `func ShutdownNode(node *Node)`
- `finger_table.go`
  - `func (node *Node) initFingerTable()`
  - `func (node *Node) fixNextFinger(ticker *time.Ticker)`
  - `func fingerMath(n []byte, i int, m int) []byte`
- `kv_store.go`
  - `func Get(node *Node, key string) (string, error)`
  - `func Put(node *Node, key string, value string) error`
  - `func (node *Node) locate(key string) (*RemoteNode, error)`
  - `func (node *Node) GetLocal(req *KeyValueReq) (*KeyValueReply, error)`
  - `func (node *Node) PutLocal(req *KeyValueReq (*KeyValueReply, error)`

  - `func (node *Node) TransferKeys(req *TransferReq) (*RpcOkay, error)`

- `node_rpc_impl.go`
  - `func (node *Node) GetSuccessorId(req *RemoteId) (*IdReply, error)`
  - `func (node *Node) SetPredecessorId(req *UpdateReq) (*RpcOkay, error)`
  - `func (node *Node) SetSuccessorId(req *UpdateReq) (*RpcOkay, error)`
  - `func (node *Node) FindSuccessor(query *RemoteQuery) (*IdReply, error)`
  - `func (node *Node) Notify(remoteNode *RemoteNode) (*RpcOkay, error)`
  - `func (node *Node) ClosestPrecedingFinger(query *RemoteQuery) (*IdReply, error)`
- `util.go`
  - `func Between(nodeX, nodeA, nodeB []byte) bool`
  - `func BetweenRightIncl(nodeX, nodeA, nodeB []byte) bool`

The difficulty in this project does not lie in the quantity of code you will write. Understanding how Chord works and protecting code that may be run concurrently are the biggest hurdles to completing a correct implementation.

4 Demo

A TA implementation of Chord is available at

`/course/cs138/pub/chord/{linux,darwin,windows}/chord`

Note that the demo is built with an m value of 16 bits, and is not fault tolerant.

Once your implementation is sufficiently functional, you should test with the TA implementation for interoperability.

5 Testing

We expect to see several good test cases. This is going to be worth a portion of your grade. Exhaustive Go tests are sufficient. You can check your test coverage by using Go's coverage tool[2].

a. **cli.go** - This is a Go program that serves as a console for interacting with Chord, creating nodes and querying state on the local node(s). We have kept the CLI simple but you are welcome to improve it as you see fit.

   You can pass the following arguments to cli.go (no arguments implies the first node)

   - -id="": ID of a node in the Chord ring you wish to join
   - -addr="": Address of a node in the Chord ring you wish to join

[2] `http://blog.golang.org/cover`

- -count=1: Total number of Chord nodes to start up in this process

You get the following set of commands available to you in the terminal:

- node
  - Display node ID, successor, and predecessor
- table
  - Display finger table information for node(s)
- addr
  - Display node listener address(es)
- data
  - Display datastore(s) for node(s)
- get <key>
  - Get value from Chord ring associated with this key
- put <key> <value>
  - Put key/value into Chord ring
- debug on|off
  - Toggle debug statements on or off
- quit
  - Quit node(s)

b. **chord/simple_test.go** - Contains a simple example test that you can use as a starting point. This test simply attempts to create a single Chord node and fails if an error is returned by the CreateNode call.

c. You're encouraged but not required to write client applications, using the `Node` struct and its provided methods, to test your implementation.

d. Go provides a tool for detecting race conditions[3], which can be helpful for detecting and debugging concurrency issues. To run all Go tests in your current directory with the race detector on, run:

```
go test -race
```

Note that you can also use the race detector while building or running your Go programs.

6 Style

CS 138 does not have an official style guide, but you should reference "Effective Go" for best practices and style for using Go's various language constructs.

Note that naming conventions in Go can be especially important, as using an upper or lower case letter for a method name affects the method's visibility outside of its package.

At a minimum, you should use Go's formatting tool `gofmt` to format your code before handing in.

You can format your code by running:

[3] `http://blog.golang.org/race-detector`

```
gofmt -w=true *.go
```

This will overwrite your code with a formatted version of it for all go files in the current directory.

7 Getting Started

Before you get started, please make sure you have read over, understand, and have set up all the common code.

To get started, run the following command:

```
go get github.com/brown-csci1380/<team-name>
```

where `<team-name>` is the name of the team repository created for you and your partner by the course staff. This command will download the stencil code for Chord.

Next, navigate to the directory where the stencil code was pulled to (this is determined by the `$GOPATH` environment variable. For more information on setting up Go, check out our "Get Going with Go" guide!) and run the following command:

```
go get ./...
```

This will pull in all of the imports from the current package downwards.

8 Handing in

You should write a simple README, documenting the Go tests you wrote and other ways you tested your project (if applicable), any bugs you have in your code, extra features you added (including extra credit features), and anything else you think the TAs should know about your project. Once you have completed your README and project, you should hand in your `chord` by running the electronic handin script

```
/course/cs138/bin/cs138_handin chord
```

to deliver us a copy of your code.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS138 document by filling out the anonymous feedback form:

`http://cs.brown.edu/courses/cs138/s17/feedback.html`.