CS 3410: Chord Distributed Hash Table

Introduction

In this assignment, you will implement a basic CHORD distributed hash table (DHT) as described in this paper:

- https://pdos.csail.mit.edu/papers/ton:chord/paper-ton.pdf

You can download my Linux implementation to play with. Download it using:

```
curl -so chord-example https://cit.dixie.edu/cs/3410/chord-example
```

Make it executable:

```
chmod 755 chord-example
```

Then launch it using:

```
./chord-example
```

Requirements

Your DHT must implement the services to maintain a DHT, and it must support a simple command-line user interface. Each instance (running on different machines, or on the same machine on different ports) will run an RPC server and act as an RPC client. In addition, several background tasks will run to keep the DHT data structures up-to-date.

User interface

When running, an instance of your DHT should support a simple command-line interface. It will supply a prompt, and the user can issue one of the following commands. The simplest command is:

- `help`: this displays a list of recognized commands

The main commands start with those related to joining and leaving DHT rings:

- `port <n>`: set the port that this node should listen on. By default, this should be port 3410, but users can set it to something else.

  This command only works before a ring has been created or joined. After that point, trying to issue this command is an error.

- `create`: create a new ring.

  This command only works before a ring has been created or joined. After that point, trying to issue this command is an error.

- `join <address>`: join an existing ring, one of whose nodes is at the address specified.

  This command only works before a ring has been created or joined. After that point, trying to issue this command is an error.

- `quit`: shut down. This quits and ends the program. If this was the last instance in a ring, the ring is effectively shut down.

  If this is not the last instance, it should send all of its data to its immediate successor before quitting. Other than that, it is not necessary to notify the rest of the ring when a node shuts down.

Next, there are those related to finding and inserting keys and values. A `<key>` is any sequence of one or more non-space characters, as is a value.

- `put <key> <value>`: insert the given key and value into the currently active ring. The instance must find the peer that is responsible for the given key using a DHT lookup operation, then contact that host directly and send it the key and value to be stored.

- `putrandom <n>`: randomly generate *n* keys (and accompanying values) and put each pair into the ring.

Useful for debugging.

- `get <key>`: find the given key in the currently active ring. The instance must find the peer that is responsible for the given key using a DHT lookup operation, then contact that host directly and retrieve the value and display it to the local user.
- `delete <key>`: similar to lookup, but instead of retrieving the value and displaying it, the peer deletes it from the ring.

Finally, there are operations that are useful mainly for debugging:

- `dump`: display information about the current node, including the range of keys it is resposible for, its predecessor and successor links, its finger table, and the actual key/value pairs that it stores.
- `dumpkey <key>`: similar to `dump`, but this one finds the node resposible for `<key>`, asks it for its dump info, and displays it to the local user. This allows a user at one terminal to query any part of the ring.
- `dumpaddr <address>`: similar to above, but query a specific host and dump its info.
- `dumpall`: walk around the ring, dumping all information about every peer in the ring in clockwise order (display the current host, then its successor, etc).

Of these, `dump` is the most helpful and is required. The others may prove helpful in debugging, but they are optional.

DHT interface

When communicating with other DHT nodes, your DHT should use the basic operations described in the paper, including finger tables. You should implement a successor list as described in the paper, and basic migration of key/value pairs as nodes join and leave. Details are given later in this document.

There are a few different basic types of values that you will need to work with. For node addresses, use a string representation of the address in the form `address:port`, where address is a dotted-decimal IP address (not a host name). When referring to positions on the ring (ring addresses), you will be using a large number, not a string. This works out nicely: keys and node addresses are strings, and both can be hashed into ring addresses, which are not strings. The type checker will make sure you never mix up the two types of addresses.

You will use the sha1 hash algorithm (available in `crypto/sha1` in the Go standard library), which gives 160-bit results. This means that your finger table will have up to 160 entries. In order to treat these as large integer operations and perform math operations on them, you will need to use the `math/big` package. I recommend immediately converting sha1 hash results into `big` values and using that as your type for ring addresses.

Each node will need to maintain the following data structures as described in the paper:

- `finger`: a list of addresses of nodes on the circle. Create a list with 161 entries, all of which start out empty (`None`). You will only need to maintain a few of them at the end, but by creating a full-size list you will simplify your code and make it as similar to the pseudo-code in the paper as possible. Use entries 1–160 (instead of 0–159) for the same reason.
- `successor`: a list of addresses to the next several nodes on the ring. Set the size of this list as a global, named constant with a value of at least 3.
- `predecessor`: a link to the previous node on the circle (an address)

Note that addresses should contain an IP address and a port number.

The main operations you must support are:

- `find_successor(id)`: note that this is not the same as the `get` function from the command-line interface. This is the operation defined in Figure 5 of the paper.
- `create()`: note that this is not the same as the `create` function in the application interface; it is the operation defined in Figure 6 of the paper (and is invoked by the command-line interface).
- `join(n')`: note that this is not the same as the `join` function in the application interface.
- `stabilize()`
- `notify(n')`

- `fix_fingers()`
- `check_predecessor()`

Each of these is described in the pseudo-code in the paper. You must also incorporate the modifications described in the paper to maintain a list of successor links instead of a single successor value.

Suggestions

The following are implementation suggestions, which you are free to follow or ignore.

Goroutines

- When you start the DHT (either through a create request or a join request), you should launch the RPC server for the local node. This is similar to what you have already done in the first project. Set up the necessary data structures, then launch the RPC server in its own goroutine.
- Run the stabilize procedure in its own goroutine, also launched at node creation time. It should run in a forever loop: sleep for a second, then call the stabilize procedure described in the paper.
- Similarly, fixfingers should run every second or so in its own goroutine.
- CheckPredecessor also runs in its own goroutine, also in a loop that repeats every second or so.
- Use the main goroutine to run the command-line interface loop. When it exits, everything else will be shut down as well.

RPC connections

The number and locations of peers will vary over time. For a production system, you would maintain a pool of outgoing connections and garbage collect connections over time.

To make things simpler, establish a fresh RPC connection for each message you send, wait for the response, then shut down that connection. You may find it helpful to write a function like this:

```
func call(address string, method string, request interface{}, reply interface{}) error
```

This function takes care of establishing a connection to the given address, sending a request to the given method with the given parameters, then closes the client connection and returns the result.

It is okay to make all of your requests synchronously, i.e., the goroutine that sends the request can stop and wait until the response is available.

Iterative lookups

Use the iterative style of recursive lookups as described in the paper. All RPC calls will be able to return values immediately without blocking, i.e., every RPC server function queries or modifies local data structures without having to contact other servers.

The most complicated operation you must support is a complete `find_successor` lookup that may have to contact multiple servers. It should start by checking if the result can be found locally. If not, it should enter a loop. In each iteration, it contacts a single server, asking it for the result. The server returns either the result itself, or a forwarding address of the node that should be contacted next.

Put in a hard-coded limit on the number of requests that a single lookup can generate, just in case. Define this as a global, named constant. 32 ought to be sufficient for hundreds of nodes.

Here is revised pseudo-code for `find_successor` and friends:

```
    // ask node n to find the successor of id
    // or a better node to continue the search with
    n.find_successor(id)
        if (id ∈ (n, successor])
            return true, successor;
        else
            return false, closest_preceding_node(id);

    // search the local table for the highest predecessor of id
    n.closest_preceding_node(id)
```

```
        // skip this loop if you do not have finger tables implemented yet
        for i = m downto 1
            if (finger[i] ∈ (n,id])
                return finger[i];
        return successor;

    // find the successor of id
    find(id, start)
        found, nextNode = false, start;
        i = 0
        while not found and i < maxSteps
            found, nextNode = nextNode.find_successor(id);
            i += 1
        if found
            return nextNode;
        else
            report error;
```

Hash functions

To get a sha1 hash value, include the appropriate libraries and use something like this:

```
func hashString(elt string) *big.Int {
    hasher := sha1.New()
    hasher.Write([]byte(elt))
    return new(big.Int).SetBytes(hasher.Sum(nil))
}
```

Now you can use the operations in `math/big` to work with these 160-bit values. It is a bit cumbersome because you cannot use infix operators. For example, the following is helpful:

```
const keySize = sha1.Size * 8
var two = big.NewInt(2)
var hashMod = new(big.Int).Exp(big.NewInt(2), big.NewInt(keySize), nil)

func jump(address string, fingerentry int) *big.Int {
    n := hashString(address)
    fingerentryminus1 := big.NewInt(int64(fingerentry) - 1)
    jump := new(big.Int).Exp(two, fingerentryminus1, nil)
    sum := new(big.Int).Add(n, jump)

    return new(big.Int).Mod(sum, hashMod)
}
```

This computes the address of a position across the ring that should be pointed to by the given finger table entry (using 1-based numbering).

Another useful function is this:

```
func between(start, elt, end *big.Int, inclusive bool) bool {
    if end.Cmp(start) > 0 {
        return (start.Cmp(elt) < 0 && elt.Cmp(end) < 0) || (inclusive && elt.Cmp(end) == 0)
    } else {
        return start.Cmp(elt) < 0 || elt.Cmp(end) < 0 || (inclusive && elt.Cmp(end) == 0)
    }
}
```

This one returns true if `elt` is between `start` and `end` on the ring, accounting for the boundary where the ring loops back on itself. If inclusive is true, it tests if `elt` is in `(start,end]`, otherwise it tests for `(start,end)`.

It is helpful to be able to print out hash codes in a format that makes it easy to visually compare two hash values. The following code will do this:

```
hex := fmt.Sprintf("%040x", hashString(value))
s := hex[:8] + ".. (" + string(value) + ")"
```

Big integers can be printed like any other integer value using variants of Printf, so this code prints is as a hexadecimal value (`%x`), padding it to be at least 40 characters long with leading zeros (the `040` between `%` and `x`). Then it takes a slice of that string to grab the first 8 characters (since that is usually plenty) and appends the original string. This is useful for keys and addresses.

Getting your network address

It is helpful to have your code find its own address. The following code will help:

```
func getLocalAddress() string {
    conn, err := net.Dial("udp", "8.8.8.8:80")
    if err != nil {
        log.Fatal(err)
    }
    defer conn.Close()

    localAddr := conn.LocalAddr().(*net.UDPAddr)

    return localAddr.IP.String()
}
```

Implementation order

Week 1

- Start by building your keyboard handler. Implement simple functions like `help`, `quit`, `port`, etc.

- Implement a skeleton `create` function that starts a `Node` instance complete with RPC server. Start by having it listen for a `ping` method that just responds to all requests immediately.

- Implement a `ping` command for the keyboard, which issues a ping request to a given address (as explained earlier). Use this to test if your RPC setup is working. You should use the `call` function given earlier. This will give you a complete RPC client/server pair. Note: `ping` will not be part of the final Chord ring, but it is a simple way to test that your RPC connections are working.

- Implement `get`, `put`, and `delete` RPC server functions that manipulate the bucket of key/values pairs stored on a given instance. The node should maintain a `map[string]string` holding the values on the local node, and the `get`, `put`, and `delete` methods modify that local `map`.

  Implement `get`, `put`, and `delete` keyboard commands that require an address to be specified (later you will find the address by invoking your find function). These are a partial implementation of what you will eventually need. In the final version, the user would type:

  ```
  put state utah
  ```

  to trigger sending a `put` RPC request to the appropriate remote node. The implementation would start by looking up the node responsible for the key “state” and then issue the appropriate RPC call. In this partial version, the call would look like:

  ```
  put state utah 144.38.198.13:3410
  ```

  The implementation skips the lookup step and uses the supplied address instead. This lets you create the UI and test the RPC functionality before you have implemented the Chord protocol.

  Test everything with multiple instances running. You should be able to insert keys into remote instances, retrieve them, and delete them.

- Start implementing your `dump` command, which will initially just list the key/value pairs stored locally.

- Make a ring. Start by ignoring finger tables, and make a ring that just uses successor pointers. Follow the rules outlined in the paper. Implement the `join` command that initially just sets the given address as your successor. An example use of the command would be:

  ```
  join 144.38.198.13:3410
  ```

  In the final version, the given address will be used as the starting point in a lookup to find this node’s successor. In this version, the local node uses the given address as the successor. In order to build a ring with multiple nodes, you will need to manually figure out the ring order of the nodes and use “join” to build them in the correct sequence.

  Note: you will actually need to track a list of successors, so at this point you should store the successor link as a slice of size one to make it easier to update later.

- Update `dump` to show all interesting information about this node.

Week 2

- Implement a timer event for the `stabilize` function. This will require implementing `notify` as well. You goal is to allow a complete ring with valid successor and predecessor pointers. Each node will maintain a list of successors as described in the later part of the Chord paper.

  You should store the successor list and the predecessor link as fields of your Node object, the same one you are publishing over RPC. Whenever you need the immediate successor, you should access the first element in the successor list.

  The pseudocode for `stabilize` calls for looking up the successor's predecessor link. Add an RPC method that a node can call to get this information from its successor. In addition to the predecessor address, the call should return the list of successors. Create a global constant that controls the number of successor links you will track. Three or four is a reasonable size.

  When the successor node returns its list of successors, you should update your successor list as follows:

  - Add your successor's address to the beginning of the list you got from your successor

  - Truncate the list if needed so it is not too long (measuring it against the maximum length discussed above).

  - Replace your old successor list with this new one

  When `stabilize` tries to contact the successor and it fails for some reason, assume that your successor has gone down. Chop the first element off your successors list, and set your successor to the next element in the list. If there is no such element (the list is empty), set your successor to your own address.

  This management of a list of successors will allow you to heal the ring after a node failure.

- Implement the initial version of the find function. Have it follow the list of successor pointers (since no finger table exists yet). You should implement `find` using the pseudocode given earlier. `find_successor` should be this revised version for now:

  ```
  n.find_successor(id)
      if (id ∈ (n, successor])
          return true, successor;
      else
          return false, successor;
  ```

  This skips the call to `closest_preceding_node` and ignores finger tables, so every lookup ends up following the linked list of nodes around the circle.

  This should give you correct lookups, they are just less efficient than the versions that use finger tables.

- Revise your "join" command to use `find`. It should use the node supplied by the user as the starting point for a lookup, and should look up its own address. The result will be its true successor, which it can then use to put itself at the correct point on the ring.

  The node should not start running `stabilize` until it has discovered its correct point on the ring.

- Update `get`, `put`, and `delete` to use your find function to find the address where a key/value pair should be located instead of requiring the user to specify it.

- Make two new RPC server functions. `put_all` receives a map of key/value pairs, and adds all of its contents to the local bucket of key/value pairs.

  `get_all` takes the address of a new node that is between you and your predecessor. It should gather all keys that belong to that new node (use your `between` function to determine this) into a new map, and it should also remove them from your bucket. You can loop through all the values in a map like this:

  ```
  for key, value := range elt.Bucket { ... }
  ```

  Return the map of values that no longer belong in your node.

- When a node is about to go down (in response to a `quit` command, call `put_all` on its successor, handing it the entire local bucket before shutting down.

- When joining an existing ring, issue a `get_all` request to your new successor once the join has succeeded, i.e., as soon as you know your successor.

- At this point, you should be able to build rings with all basic functionality in place. The rest is just optimization.

- Add `fix_fingers` and `check_predecessor` and update `stabilize` to conform to the pseudo-code in the paper (if it does not already).

  When `fix_fingers` finds a entry, add it to the next entry (as given in the pseudo-code), but also loop and keep adding to successive entries as long as it is still in the right range. For example, the successor for the position 1 step further around the ring is probably the successor for 2 steps around the ring, and for 4, 8, and 16 steps around the ring. You can check this without repeating the `find_successor` operation, and will likely fill the first 155 or so of the 160 entries with a single value. This is a good optimization.

- Update your `find_successor` function to use the finger table. In `closest_preceding_node`, the last line (the fallback case if nothing suitable was found in the finger table) should return your successor, not your own address.

At this point, you should have complete functionality.