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://www.cs.utahtech.edu/cs/3410/chord-example
```

Make it executable:

```
chmod 755 chord-example
```

Then launch it using:

```
./chord-example
```

Starter code is available here:

- https://www.cs.utahtech.edu/cs/3410/chord-starter/

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, the following operation is useful mainly for testing:

- `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.

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 slice with 161 entries, all of which start out empty. Use entries 1–160 (instead of 0–159) to simply and make your code as similar to the pseudo-code in the paper as possible.
- `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 adapt.

Node

A node on a Chord ring can be represented by an object. The following should suffice (along with a few other helpful types):

```
type Key string

type NodeAddress string

type Node struct {
    Address     NodeAddress
    FingerTable []NodeAddress
    Predecessor NodeAddress
    Successors  []NodeAddress

    Bucket map[Key]string
}
```

Both keys and node addresses are represented as strings, but it can be helpful to define distinct types for them so the compiler can help you avoid mixing them up unintentionally. That also lets you define methods on those types, for example a helper method that hashes the key/address.

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. Set up the necessary data structures, register the Node object with the RPC server library, then create a goroutine to listen for incoming requests. Note that this goroutine should only be created in response to a create or join request from the user.

- Create a single goroutine to run the background maintenance procedures in a loop. Run each of the following procedures in a loop with a delay of $^1⁄_3$ of a second between each, so each will end up being called about once per second:

    - Stabilize
    - FixFingers
    - CheckPredecessor

  This goroutine should only be created in response to a create or join request from the user.

- Use the main goroutine to run the command-line interface loop. When it exits, everything else will be shut down as well.

  When the process is first launched, no background goroutines will be running and incoming requests will not be accepted. The text-based shell will be the only thing running initially so the user can set up the node and then run a create or join command.

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 lookup (which we will call `find`) that may have to contact multiple servers. It should run a loop. In each iteration, it contacts a single server (in many cases it will be contacting itself via RPC, but this is okay), 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()
}
```

Finding your local IP address is suprisingly hard to do reliably, in part because a machine usually has multiple addresses. This code prepares (but does not send) a UDP message to a well-known IP address (Google’s public DNS service) and then looks up the return address that the system assigns to the message. This is usually the a good address for contacting your machine, assuming it is publicly routable (or at least routable within your current network).