Project Guide for Kademlia

Building

Go’s build tools depend on the value of the GOPATH environment variable. $GOPATH should be the project root: the absolute path of the directory containing {bin,pkg,src}.

Once you’ve set that, you should be able to build the skeleton and create an executable at bin/main with:

    go install main

Running main as

    main localhost:7890 localhost:7890

will cause it to start up a server bound to localhost:7890 (the first argument) and then connect as a client to itself (the second argument). All it does by default is perform a PING RPC and exit.

Skeleton Code

The skeleton code is provided in three files: main.go in the main package, kademlia.go, rpc.go and id.go in the kademlia package. The core functions within Kademlia such as the k-buckets, update, and protocol functions should go in the kademlia.go file. RPC responses which are invoked by remote machines are placed in the rpc.go file. It may be helpful to create a new file, kbucket.go, to represent each k-bucket and implement helper functions such as MoveToTail(), etc...

Plan of Attack

1. Implement routing tables with K-Buckets.

Routing in Kademlia is done through the use of distance-based collections of Contacts, triplets containing a node’s key, ip and port. These collections are known in Kademlia as k-buckets, structures which hold at most k contacts.

Since we are using 160 bit keys for node ids in our implementation, there should be 160 k-buckets in your kademlia object. Each found contact should
be placed in the bucket corresponding to its distance using the formula below (where distance is an XOR between the two keys):

\[ 2^j \leq \text{distance}(\text{node}, \text{contact}) < 2^{(j+1)} \]

K-buckets are sorted by time of most recent contact (most recent at the tail of the k-bucket). Whenever your Kademlia successfully communicates with any node in the DHT, it updates (calling the Update() function) the node in the corresponding k-bucket for the following conditions:

- If the contact **exists**, move the contact to the end of the k-bucket.
- If the contact **does not exist** and the k-bucket is **not full**: create a new contact for the node and place at the tail of the k-bucket.
- If the contact **does not exist** and the k-bucket is **full**: ping the least recently contacted node (at the head of the k-bucket), if that contact fails to respond, drop it and append new contact to tail, otherwise ignore the new contact and update least recently seen contact.

2. **Implement basic RPCs for Ping(), Store(), FindNode() and FindValue()**

If you have not done so already, please read about RPCs and GO here:

[http://jan.newmarch.name/go/rpc/chapter-rpc.html](http://jan.newmarch.name/go/rpc/chapter-rpc.html)

As a refresher, all RPCs in GO are an interface with three restrictions. The function must be public (begin with a capital letter), have exactly two arguments, the first is a pointer to value data to be received by the function from the client, and the second is a pointer to hold the answers to be returned to the client, have a return value of type error.

In order to use RPC methods, you have to register a struct and accompanying methods with the global RPC module, then launch an RPC serving handler. Luckily this is already handled for you in the main() method in the skeleton code.

The RPCs needed to implement the Kademlia protocol are the four given above. These RPCs are invoked remotely from other Kademlia nodes (hence the Remote procedure call in RPC). Remember that the return values of Go RPCs is the second argument in the function in addition to the error object.

In addition to the RPC handlers in rpc.go, you will need client versions of these methods which invoke RPCs in remote nodes and implement the logic of the Kademlia protocol. The four functions you will need to implement in kademlia.go are below:

```
DoPing(remoteHost net.IP, port uint16)
```
The Ping is the primitive communication for node organization and update. A node sends out a ping request to other nodes. When a node receives a Ping message, it responds with a Pong message. Ping and Pong structs are provided in the skeleton in rpc.go.

**DoStore(remoteContact *Contact, Key ID, Value []byte)**

Store() is the primitive for storing a key-value pair remotely in the DHT. The Store() primitive takes three arguments, a Contact who should store the value, an ID for the key, and a byte array for the value.

**DoFindValue(remoteContact *Contact, Key ID)**

The FindValue RPC includes a B=160-bit key. If a corresponding value is present on the recipient, the associated data is returned. Otherwise the RPC is equivalent to a FIND_NODE and a set of k triples is returned.

**DoFindNode(remoteContact *Contact, searchKey ID)**

The FIND_NODE RPC includes a 160-bit key. The recipient of the RPC returns up to k triples (IP address, port, nodeID) for the contacts that it knows to be closest to the key.

The recipient must return k triples if at all possible. It may only return fewer than k if it is returning all of the contacts that it has knowledge of.

3. **Implement iterative versions of RPCs**

The iterative versions of the RPCs in the protocol should be done last, after the basic versions of the RPCs have been thoroughly tested, since they build on them.

Three iterative RPCs should be implemented using the logic in the iterative node lookup: iterativeStore(), iterativeFindNode(), iterativeFindValue().

**iterativeFindNode()**

The iterativeFindNode() function forms the basics for node lookup within Kademlia. For each iterative cycle of RPCs, the client collects a list of k closest contacts, obtained by iteratively querying nodes from its own k-buckets, and those contacts returned from previous RPC iterations (i.e. nodes returned from FIND_NODE).

Each iteration, known as a cycle, consists of a set of alpha (in our case 3) RPCs being sent out and fully returned or timed out. The function first selects 3 nodes from it's own k-buckets which are closest to the key. If there
are not at least 3 contacts in the correct k-bucket, use from the next closest and so on. If there are fewer than 3 contacts, start with only those, but continue to use 3 once more have been returned by remote nodes in subsequent iterations.

The Contacts are stored in a list known as the shortlist, and should at the end contain the k closest contacts from a combination of the client's own k-buckets and the contacts returned from queried clients. In addition to the Contact, there should also be a flag for each contact to denote whether the node is active, and has responded to that function's RPC call. Within function, the closest node in the shortlist is tracked.

During each cycle, the Kademlia object launches 3 concurrent FIND_NODE RPCs to the Contacts in its shortlist which have not been contacted during this call to iterativeFindNode. When each RPC returns, the nodes given by that node are added to the shortlist if they do not currently exist. The Contact probed is marked as active within the shortlist.

Since Kademlia uses loose parallelism, the RPC cycles are sent out after a specified time interval, leading there to potentially be more than alpha contacts in flight. This is fine, and for our purposes, use a time interval of 300 milliseconds; however, this is not an absolute. Feel free to play around with this interval to improve performance.

The iterations continues until either (1) no Contacts returned are closer than currently exist in the shortlist, or (2) there are k active contacts in the shortlist.

A set of k triples is returned from this function.

**iterativeStore()**

This function first calls iterativeFindNode() and receives a set of k triples. A STORE rpc is sent to each of these Contacts.

**iterativeFindValue()**

This function is very similar to the iterativeFindNode described above, except that it uses the FIND_VALUE RPC instead of FIND_NODE. Also, the function terminates if at any point the value is returned instead of a set of Contacts (i.e. the value is found), in addition to the termination constraints of iterativeFindNode.

When this function succeeds (finds the value), a STORE RPC is sent to the closest Contact which did not return the value.
Command-Line Interface

As demonstrated above, your program must accept two positional arguments of the form "host:port". The first tells it the bind address of its own server; the second gives the first peer your client should connect to to join the network.

After setting up its server and establishing a connection to its first peer, your executable should loop forever, reading commands from stdin, executing them, and printing their results to stdout. All data should be printed with the %v specifier and should be followed by exactly one newline. You may assume values are alphanumeric and are no more than 4095 B. All operations should complete within 10 seconds.

Implement the following commands:

whoami
  Print your node ID.

local_find_value [key]
  If your node has data for the given key, print it.
  If your node does not have data for the given key, you should print "ERR".

get_contact [ID]
  If your buckets contain a node with the given ID,
    printf("%v %v\n", theNode.addr, theNode.port)
  If your buckets do not contain any such node, print "ERR".

iterativeStore [key] [value]
  Perform the iterativeStore operation and then print the ID of the node that received the final STORE operation.

iterativeFindNode [ID]
  Print a list of ≤ k closest nodes and print their IDs. You should collect the IDs in a slice and print that.

iterativeFindValue [key]
  printf("%v %v\n", ID, value), where ID refers to the node that finally returned the value. If you do not find a value, print "ERR".

// The following four commands cause your code to invoke the appropriate RPC on // another node, specified by the nodeID argument.

ping [nodeID]
ping [host:port]
  Perform a ping.

store [nodeID] key value
Perform a store and print a blank line.

find_node [nodeID] [key]
    Perform a find_node and print its results as for iterativeFindNode.

find_value [nodeID] [key]
    Perform a find_value. If it returns nodes, print them as for find_node. If it returns a value, print the value as in iterativeFindValue.