Programming Assignment 2: Distributed Hash Table
Part 1: Routing

CSE 486/586: Distributed Systems

Introduction

For this project, you will be implementing a routing table for a distributed data structure called k-DHT that is *similar to* the distributed hash table Kademlia [1]. It is less robust and less sophisticated than the full Kademlia DHT protocol, but it will serve to deepen your understanding of distributed hash tables, distributed data structures and communication patterns in general, and peer-to-peer architectures. Think of it as Kademlia, the Polex to Kademlia's Rolex. It turns your wrist a little bit green if you wear it all day, but it was only $20 and from across the street it looks pretty suave.

In Part 1 of this project you will learn about simple content-based addressing methods, and distributed routing. You will use cryptographic primitives, and more complex Go interfaces, more complex data structures than we have seen in previous pr

Part 2 of this project   uted hash table. Because this would otherwise cr   ou will be provided with a working routing table   plementation is not fully functional. Note that th   osts (this should include WSL2 on Windows).
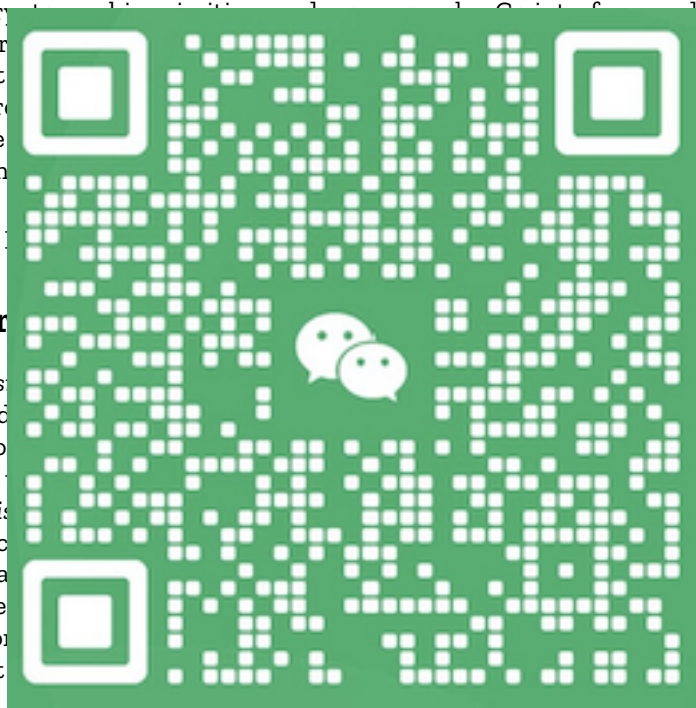
This is a three week

Academic Integr

This is an *individual ass*   in accordance with the course, department, and   g code or specific design decisions is a violation o   other Internet site. The allowable resources for   nd protobuf.dev, lecture notes, the course text *Dis*   d optional readings from this handout and the lec

Remember that it's a

For this project, note   *a, and implementations of* *Kademlia in Go.* Using, or   s a violation of academic integrity and will result

1   Getting Started

First, read *Kademlia: A Peer-to-peer Information System based on the XOR Metric* [1]. Pay special attention to Section 2 of that document, which describes the routing table structure and the interactions of the data structure as nodes are added and removed.

Next, read this handout and all of the given code carefully. The given code contains numerous comments and some tests that will help you understand the requirements of this project. In particular, the tests in the directory tests/ in the given code are tests for the public API of the routing table, and demonstrate how it will be used.

Pay special attention to the figures in this section, as they will help you understand the structure of the routing table. Note that, while the routing table is repeatedly described as a tree (and it is a tree!), it can also be thought of as a particular division of the key space into logarithmically decreasing partitions surrounding the local node's own key. At each partitioning, the remaining key space is divided in half, until the node is alone in its own "bucket" of the key space. While this forms a tree, it is a tree with nodes representing non-overlapping

portions of the key space, and so you will probably want to represent it in your implementation as an array or list. Keep this in mind when reading the paper, and it may help to understand the requirements.

Notice that some data structures used in the routing table (such as `kdht.NodeInfo`) are defined in `messages.proto` and require building the Protobuf definitions. Running `make` in the top-level directory of the given code should build the Protobuf implementation (as well as update `go.sum`).

2 Requirements

You must implement the `api/kdht.RoutingTable` API in the given code of this project. Note that the given code includes another API, `api/kdht.Node`, which *you will not need to implement yet*. You may find it helpful to read the comments for that API to put your routing table in context, however.

You *must* use the same version of Go and the Protobuf external package for this project as you did for the previous project. You may not use any other external packages, although any package in the Go standard libraries is acceptable. (This corresponds approximately to allowing any package that *does not* begin with a domain name.)

Your implementation of the API will conform to the description of the Kademlia API in the original Kademlia paper [1], with a few sim[...] ifficult to understand at first, but a few read-thro[...] ill make things clearer.

There are no specific [...] outing table or the data structures that you use, [...] particular, if a very large number of nodes are pla[...] e to find any given node in the table in time log[...] s requirement, you may find that some tests tim[...] nalysis of algorithms or data structures is requir[...] he Kademlia paper, your implementation's perfor[...]

Your implementation [...] (`os.Stdout` in Go, used by default for certain outp[...]

2.1 API

The entry point to your [...] wing function signature and requirements:

- `func NewRoutingTabl[...]` unction must create an instance of some s[...] fined below). The given node's Id field is t[...] resents the local k-DHT node. The parame[...] he maximum number of nodes placed in an[...]

The `api/kdht.RoutingTable` interface includes the following methods:

- `func (rt *RoutingTable)K()int`: This function simply returns the value k that was passed to `NewRoutingTable()` at the time that the routing table was created.

- `func (rt *RoutingTable)InsertNode(node *NodeInfo)`: This method *must* attempt to insert a node into the routing table, splitting *k*-buckets if necessary to do so. If the *k*-bucket into which the inserted node would be placed is already full, the insertion *must* do nothing. This method therefore cannot fail.

- `func (rt *RoutingTable)RemoveNode(key []byte)error`: RemoveNode *must* remove the node specified by the given key from the routing table, if it exists in the routing table. If the specified node does not exist, or if the specified node is the local k-DHT node (which must not ever be removed), then it *must* return `kdht.InvalidNodeError`.

- `func (rt *RoutingTable)Lookup(key []byte)(node *NodeInfo, ok bool)`: This looks up a node by its key in the routing table, and *must* return the node and a true `ok` value if it exists in the routing table. If it does not exist in the routing table, it must return a false `ok` value.

- `func (rt *RoutingTable)GetNodes(bucket int)[]*NodeInfo`: This function *must* return a slice containing exactly those nodes in the numbered bucket of the routing table. The bucket numbering is as described in the Kademlia paper, with specific requirements in Section 2.3 of this document. This operation *must* always succeed and *must* simply return `nil` if the specified bucket is empty or does not yet exist.

- `func (rt *RoutingTable)ClosestK(key []byte)[]*NodeInfo`: This function *must* return the closest *k* known nodes to the specified key. It *must not* fail and *must not* return an empty slice, but it may return fewer than *k* nodes if the routing table does not contain *k* entries.

- `func (rt *RoutingTable)Buckets()int`: Buckets *must* return the number of buckets in the current routing table, according to the rules in the Kademlia paper, with the clarification that you do *not* need to ever remove or coalesce buckets if nodes are removed from the routing table.

No key or NodeInfo structure passed to your routing table will be modified after it is given to your routing table, so you may retain references to these things if you wish to do so. Such data structures *may*, however, be *used* after passing them to your routing table, and the user will expect that they have not been modified. You therefore *must not* modify any argument that is passed to your routing table by reference. You may in any case find that it is n[...]associated methods) to store this information, r[...] particular, the tests *will never assume* that the no[...]es that were passed into `InsertNode()`, so you *may* [...]

No node should ever [...]peration.

2.2 Key Space

The key space in your i[...]ided for you in `api/kdht`. The key values provided [...]ntegers for ordering and comparison purposes. Y[...] function maintain this requirement without an[...]

2.3 Bucket Numbe[...]

Buckets are numbered f[...]may exist. (The rules for this are described later.)

- Bucket 0 represen[...]er in the *least significant bit* of the *least sign[...] to the local node by the XOR metric.

- Each bucket $i > $ [...] the local k-DHT node's identifier; that is, all bits from the most significant bit to the *i*th least significant bit of the identifier of the nodes in bucket *i* are the same as the corresponding bits in the local k-DHT node's identifier. Thus bucket 159 is the "other half" of the key space from the local node, bucket 158 is one quarter of the key space that has the same most significant bit as the local node but a different second most significant bit, *etc*.

- Nodes in the *lowest-numbered non-empty bucket* may have *either bit value* in their corresponding bit position. For example, if there are 32 buckets (that is, bucket number 127 is the last non-empty bucket), every node in buckets $128 <= i < 159$ shares an identifier prefix of *exactly* $159 - i$ bits with the local k-DHT identifier, and has the *opposite bit value* in the $159 - i$st most significant (counting from zero) bit. The nodes in bucket 127 share a 31-bit prefix with the local k-DHT node, but the 32nd bit may have *either bit value*.

Note that unless there are an enormous number of nodes in the buckets, small-numbered buckets will probably not exist, or will be empty. The index of each bucket is where nodes that differ at the identified bit *would be located if all buckets were full*, but in practice they may be in some bucket *of a larger index* when the routing table is not full.

These rules will be naturally maintained if the splitting of buckets as described in the Kademlia paper is followed.

In certain pathological circumstances, it is possible that there may be empty buckets between bucket 159 and the last-occupied bucket of the routing table. This may occur either because of non-uniform distribution of node identifiers in the key space or due to node removals. Your implementation *must not* renumber buckets or coalesce adjacent buckets to prevent this. The good news is that the required behavior is *strictly easier* than dealing with re-balancing buckets in this situation.

The Kademlia paper includes a special rule which guarantees that the *k* closest nodes to the local k-DHT node's identifier are always maintained, even if this distorts the k-bucket tree's structure. Kadernlia, and thus your implementation, does not have this feature. This makes your implementation somewhat simpler at the cost of unreliability for certain node distributions.

2.4 Assumptions

You *may* assume that all keys and node identifiers passed to your routing table are properly formed and of the required length (kdht.KeyBytes bytes, or kdht.KeyBits bits). You may wish to check this for debugging purposes, but your code will not be ⬚

It is bad style to hard⬚ ⬚ou should avoid this, but feel free to use the defi⬚ ⬚s allows you to define a routing table of fixed len⬚

2.5 Concurrency

Your routing table *must* ⬚ ⬚le any particular performance or ordering guar⬚ ⬚ simply acquire a mutex upon entering any meth⬚ ⬚ing that mutex. In Part 2 of this project you may f⬚ ⬚trol (or you may not), but you do *not* need to worry⬚

2.6 Other APIs

The given code contains ⬚ *r Part 1 of this project*. In particular, the following⬚

- The function impl.⬚
- The interface kdht.⬚
- The Protobuf enum⬚
- The Protobuf messsage Message

In addition, while the structure kdht.NodeInfo is a Protobuf message, you will only need to use its Id and Address fields for Part 1 of this project.

3 Guidance

I cannot stress enough the importance of reading and understanding the Kademlia paper. Please ask questions on Piazza or in office hours if you have them! You *will absolutely find it helpful* to draw some diagrams of the key space and how the *k*-buckets of the routing table react as nodes are added and the buckets are divided up.

3.1   Software Design

You will probably find it helpful to define several simple structures with associated methods for this project. For example:

- A routing table structure holding the metadata provided to `NewRoutingTable()` and some sub-structures.

- A *k*-bucket structure that understands when it is full, whether the nodes stored in it differ from the local k-DHT identifier in exactly the bit associated with the bucket or in some bit farther down, *etc.*

- A data structure representing a k-DHT node that holds frequently-needed metadata such as the length of its identifier's shared prefix with the local k-DHT node.

3.2   Testing

This data structure is not large or complicated in terms of its fundamental structure, but it is *quite subtle*. You will want to write additional tests for your routing table in the style of the tests in the tests/ directory in the given code, and you ma~~~~implementation that can examine its internal str~~~~progress.

3.3   Incremental D

When you begin writing~~~~ne node (the local k-DHT node given to NewRouting~~~~that you can add *k* nodes to the data structure and~~~~ming the tree structure. We will test your data st~~~~rading, so ensuring that it works well for *k* or few~~~~at least some credit. As you write and pass tests~~~~as changed if you break a test later.

4   Grading

This project is worth 10~~~~ccording to the requirements in Section 2, abov~~~~ation of this assignment will not reflect this comp~~~~re only that your project compiles and implemen~~~~have to implement your own tests!

Some requirements ~~~~not print extraneous information to standard o~~~~aintain them will cause many criteria to lose points.

The point breakdown for this project is out of 20 points, as follows.

Points	Description
1	The `K()` method functions correctly
2	The `Buckets()` method returns the correct number of *k*-buckets based on preceding operations on the routing table
2	Nodes inserted into the routing table can be looked up by their identifier
4	Nodes inserted into the routing table are inserted into the correct numbered *k*-bucket
2	Attempting to insert nodes into full *k*-buckets does not change the routing table
2	Nodes removed from the routing table no longer appear in any node retrieval operations
3	The `ClosestK()` method retrieves the closest *k* nodes to a specified identifier when its closest *k*-bucket is full
2	The `ClosestK()` method retrieves the closest *k* nodes to a specified identifier when its closest *k*-bucket is not full
2	The routing table displays reasonable performance as the number of nodes it contains grows large

References

[1] Petar Maymounkov and David Mazières. "Kademlia: A Peer-to-peer Information System based on the XOR Metric". In: *Proceedings of the International Workshop on Peer-to-Peer Systems.* Mar. 2002, pp. 53–65. URL: https://pdos.csail.mit.edu/~petar/papers/maymounkov-kademlia-lncs.pdf.