Programming Assignment 3: Distributed Hash Table
Part 2: DHT Nodes

CSE 486/586: Distributed Systems

Introduction

For this project, you will be implementing a distributed data structure called k-DHT that is *similar to* the distributed hash table Kademlia [1], which marketing has chosen to brand Kadernlia to avoid legal trouble. After proving out the routing table algorithms, you are ready to start communicating among k-DHT nodes and storing data. Since implementing all of Kademlia seems like a real bother, management has suggested that you implement only the core functionality for Kadernlia and leave things like LRU lists, failure probing, and rebalancing for after-sale support to handle, ideally after your manager has gotten their big promotion.

In Part 1 of this project you learned about simple content-based addressing methods and distributed routing. You will use a router complying with the interface specified in Part 1 to implement your Part 2.

This part of the project, Part 2, will use the routing table you implement in Part 1 in a distributed hash table. Because this would otherwise create a dependency of success in Part 1 for success in Part 2, you will be provided with a working routing table implementation that you can use for Part 2 if your Part 1 implementation is not fully functional. The implementation that you will be given will contain routers for x86-64 Linux, WSL2 on x86-64, x86-64 macOS, and Apple Silicon macOS.

This is a three week project.

Academic Integrity

This is an *individual assignment*. You are responsible for completing it on your own, in accordance with the course, department, and University Academic Integrity policies. In particular, sharing code or specific design decisions is a violation of the AI policy, as is the use of assistance from sites such as Stack Overflow or any other Internet site, or from large language models or other assistants such as Copilot or ChatGPT. The allowable resources for this project are: the documentation and tutorials on go.dev and protobuf.dev, lecture notes, the course text *Distributed Computing* by Kshemkalyani and Singhal, required and optional readings from this handout and the lecture notes, and the course Piazza instance.

Remember that it's always OK to ask a TA or an instructor!

For this project, note that *there are absolutely open source implementations of Kademlia, and implementations of Kademlia in Go.* Using, or even *looking at*, those implementations for this assignment is a violation of academic integrity and will result in failure in the course. They're not Kadernlia anyway!

1 Getting Started

You should have read *Kademlia: A Peer-to-peer Information System based on the XOR Metric* [1] (probably several times) by now. If not, read it posthaste.

The given code for this project is the code you have been using for Part 1, but there are parts of that code that you may not yet have explored thoroughly. Read the given code carefully, paying special attention to the parts that you may not have needed for Part 1.

As with Part 1, some data structures used in this project (such as `kdht.NodeInfo`) are defined in `messages.proto` and require building the Protobuf definitions.

2 Requirements

You must implement a k-DHT node, its communication mechanisms, and its data storage via the `api/kdht.Node` API for this project.

Your implementation will be a simplified Kademlia that does not handle caching or rebuild routing based on node failures. You will, however, provide all of the basic API calls from the Kademlia paper, and you should use it as your baseline for implementation except for otherwise specifically noted. Your implementation will use TCP streams instead of UDP datagrams for its communication.

As in Part 1, there are no specific requirements for the techniques with which you implement your k-DHT node, as long as it conforms to the specified API. Again, like Part 1, your implementation should be reasonably efficient; if you use a reasonably efficient routing table for local routing decisions, Go maps for key-value storage, and follow the network routing rules in the Kademlia paper, you should find that your efficiency is not an issue.

Your implementation of this API *must not* produce any output on standard output (`os.Stdout` in Go, used by default for certain output operations).

2.1 API

The entry point to your API is the function `impl.NewNode()`, which has the following function signature and requirements:

- `func NewNode(key []byte, addr string, k int, alpha int, neighbors []string)(kdht.Node, error)`: This function must create an instance of some type that implements the `api/kdht.Node` interface (defined below). The arguments to this function are:
  - `key []byte`: the node ID of the node being created
  - `addr string`: the listening address of the node being created
  - `k int`: the value k for the routing tables used in this instance of the k-DHT
  - `alpha int`: this corresponds to α from the Kademlia paper (described in Section 2.3 under the discussion of FIND_NODE)
  - `neighbors []string`: a list of *socket addresses* suitable to be passed to `net.Dial()` representing neighbors

Most of these arguments are self-explanatory. The trickier arguments include α, which is not one that we discussed in detail (though it is described in the paper), that represents the number of nodes to query in each step of `FindNode()` or `FindValue()` to provide robustness against failed nodes. The purpose of `neighbors` is to provide nodes to populate this node's initial routing table; each of those neighbors can be sent a `PING` message to retrieve its node ID and then inserted into the routing table. Note that the strings passed in as neighbors are *node IP address/port number pairs*, not node IDs.

The value returned by `NewNode()` (assuming there are no initialization errors) *must* implement the `api/kdht.Node` interface, which provides the following methods:

- `Ping(id []byte, message []byte)error`: Ping causes a `PING` message to be sent to the named node ID. The response should be parsed and the pinged node placed into the local routing table if appropriate to do so.
- `Store(value []byte)error`: This stores the data in the given byte array in the k closest nodes to its hash value (as returned by `keys.Compute()`) in the hash table by performing a recursive find node and then sending `STORE` messages to those nodes.
- `FindNode(id []byte)([]*NodeInfo, error)`: Finds the k closest nodes to the given node, using the Kademlia recursive FIND_NODE algorithm and returns them; if fewer than k nodes exist, it returns all of the nodes it finds.
- `FindValue(id []byte)([]byte, NodeInfo, error)`: This recursively finds a value in the distributed data structure if it exists, and returns it.
- `Shutdown()error`: Shutdown causes the node to stop listening on its listening address and shut down all other operations. Communications in progress may terminate, but should not crash. All goroutines associated with the node should complete their current action and then return. No new operations will be called on this data structure until `Shutdown()` has returned, and all subsequent operations should return appropriate errors.

- `Neighbors()[]NodeInfo` Neighbors should return all of the nodes that are currently known to this node's routing table, other than itself. A node that is shut down has no neighbors.

All API calls should return appropriate errors if necessary. The specific error messages appropriate for the API calls are defined in the given code.

2.2 Utilizing α

In each of the recursive search steps described by Kademlia, the searching node contacts α peers, chosen randomly from the k nodes closest to a given target. Because our implementation is not required to be robust to Byzantine failures or adversarial nodes, you *may* (and probably should) immediately use the *first* result you get back from any node. Subsequent responses from other nodes can then simply be discarded when they arrive. Your implementation *must* be prepared for any given node that it attempts to contact to be non-responsive.

2.3 Communication

Your implementation *must* use TCP to communicate with other nodes, and all messages sent by your implementation *must* be the eight message types defined in `messages.proto`: `PING`, `STORE`, `GET`, `FIND_NODE`, `FIND_VALUE`, `ACK`, `NODES`, and `VALUE`. The first five are "request" messages that request the receiving node to take some action, and the final three are "response" messages that indicate the results of a request. Each message *must* be preceded by a 16-bit unsigned big-endian integer (as in your `MessageService` implementation).

All of these messages are communicated using the *same Protobuf message type,* `api.Message`. The `Type` field of the message determines the validity and meaning of the individual fields in the Message structure and indicates which request or response it corresponds to. Every message fills out the `Sender` and `Type` fields, but all other fields have varying usefulness depending on the message type; `messages.proto` contains extensive commenting defining these relationships.

Your implementation *may* send multiple messages on the same TCP connection, and *must* be prepared to receive multiple messages on the same TCP connection. The response to a request message *must* be sent on the same TCP connection that received the request. These things should simplify your implementation.

Except as described in Section 2.8, below, there is no ordering requirement for messages which arrive on different TCP connections. However, messages which arrive on a single TCP connection *must* be processed in the order in which they are received.

The `Sender` field of every message received by your node *must* be used to update the node's routing table appropriately. In other words, if a message is received from a node which should be inserted into the routing table, that node *must* be inserted. Other incoming `NodeInfo` structures (for example, the results in a `NODES` response to a `FIND_NODE` query) *must not* be inserted (as the node may have failed!). Instead, if your node learns of a new node that *could* be inserted into the local routing table, it *must* issue a `PING` message to that node and insert it on the resulting `ACK`, if any.

2.4 Messages

Each of the request-style messages sent in this protocol produces exactly *one* message in reply, although for two of the requests there is *more than one possible reply.* Each individual request will receive exactly one of them, in this case. The pairings are as follows; note that the values in this table are *messages*, and that each message contains *data* as depicted between the brackets:

Request	$\rightarrow$	**Reply**
`PING{Value}`	$\rightarrow$	`ACK{Value}`
`STORE{Key, Value}`	$\rightarrow$	`ACK{}`
`GET{Key}`	$\rightarrow$	`VALUE{Value}` *(or)* `ACK{}`
`FIND_NODE{Key}`	$\rightarrow$	`NODES{Nodes}`
`FIND_VALUE{Key}`	$\rightarrow$	`NODES{Nodes}` *(or)* `VALUE{Value}`

More information for each of these messages is present in the `messages.proto` file, which you should read. Note that *every message uses the same Go struct*, and that the `Type` field must be used to identify the message

being sent and to figure out what kind of message you have received; each type of messages uses a different set of the other fields on the struct, as depicted in the above table and in `messages.proto`.

For the `PING` message, the value is optional, and all of my tests will ignore it and return nothing in the corresponding `Value` field of the returned `ACK`. For your own debugging purposes, you *may* find it valuable to be able to include a value in your pings, but you must not expect it.

The `STORE` message will generate an `ACK` in response, and you *must* send an `ACK` when you receive a `STORE`. You are *not* required to verify that the item *should* be stored locally (based on the DHT topology), you should just store it; therefore, stores cannot fail unless the DHT node has failed. The `Key` should be the key of the value being stored, and the `Value` should be the value to store. You *should not* check that the SHA of the value is equal to the key, just trust it. You *may* send an `ACK` containing the key and/or the value if you want, but you *must not* require this behavior.

The `GET` message is the first message we encounter which engender one of two possible replies: `VALUE`, with the `Value` being the value of the key being requested if the receiving node is *currently storing the requested key*, or `ACK` if the receiving node is *not* currently storing the requested key. Note that the receiving node *DOES NOT* attempt to contact any other node to find the requested key; if it knows the answer, it provides it, and if it does not, it acknowledges the attempt. You *may* include the key in either the `VALUE` or the `ACK`, or both, but you *must not* require this and my tests will not check it.

`FIND_NODE` messages *must* always produce a `NODES` response, which contains the $\leq k$ nodes closest to the `Key` in the `FIND_NODES` request that the receiving node is aware of in its `Nodes` field, regardless of whether *a)* the receiving node actually knows a node with that particular key, or *b)* the receiving node knows about k other DHT nodes. Note that the receiving node can always return *its own node information*, so this reply is never empty.

Finally, `FIND_VALUE` messages can also produce one of two replies: `NODES` (with similar semantics to `FIND_NODE/NODES`, described above), or `VALUE` if the receiving node is storing a value at the requested `Key`. This is essentially served as a `GET` request if `GET` would return a `VALUE` response, and as a `FIND_NODES` request if `GET` would return a `ACK` response.

None of these messages cause the receiving node to send any messages other than a single reply message in response. In particular, the receiving node *must not* attempt to contact any other node in order to serve a request message.

2.5 APIs and Iteration/Recursion

The API calls `Ping()`, `Shutdown()`, and `Neighbors()` are not iterative operations and *must not* use any information not already available at the current node. For example, if a `Ping()` operation is invoked for a node that is unknown to the local node (*i.e.*, not already stored with contact information in the local node's routing table), it *must* immediately return `InvalidNodeError` and *must not* send any `FIND_NODE` messages or begin a `FindNode()` operation.

The `FindNode()`, `FindValue()`, and `Store()` operations are iterative operations and *must* begin the recursive process of finding the target node or value, or finding the appropriate nodes on which a value should be stored, respectively. Note that the `FindNode()` method invocation *is* recursive, while the receipt of a `FIND_NODE` message is *not* recursive. Recursion is the responsibility of the node on which the Go API call is invoked, not the nodes which receive its messages. In short, `FIND_NODE` (for example) finds the closest k nodes to the specified key *already known to the local node*, while `FindNode()` attempts to find the closest k nodes to the specified key *in the entire DHT structure*.

2.6 Caching

The Kademlia paper has an extensive discussion on caching behaviors. Your implementation *must not* perform any caching. Data is stored on the local node only in response to a `STORE` message.

2.7 Assumptions

You *may* assume that all keys and node identifiers received over the network or passed in to your node functions 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 graded on its ability to handle invalid key sizes.

You may assume that no more than $\alpha - 1$ of the nodes that you contact during a node lookup and no more than $k - 1$ of the nodes closest to the target of a `FindNode()` or `FindValue()` operation have failed. This means that

you do not have to follow the routine `Ping()` and routing table node replacement rules from the Kademlia paper, which will simplify your implementation; once a node has been placed in your routing table, you will not need to remove it. You will still have to detect non-response of a node that you have queried.

2.8 Concurrency

Your node implementation *must* be capable of handling multiple concurrent operations, including both recursive operations on the local node object and network requests. You may block operations to ensure consistency within your node (for example, by locking shared data structures or blocking on channels), but you must not unnecessarily block other operations. As an example, while a node is waiting for responses from remote hosts during a recursive `FindNode()` lookup, it *must* be able to reply to an incoming `PING` or `FIND_NODE` message. On the other hand, it could reasonably block the response to an incoming `FIND_NODE` message briefly during an update to the local routing table.

There is no ordering requirement for the response to messages that arrive concurrently. However, once a reply r_1 has been sent for some message m_1, any message m_2 that arrives after the sending of r_1 must satisfy the happens before relationship $m_1 \rightarrow m_2$. For example, if a node n_0 receives a `PING` message m_1 from some node n_i and replies to that ping, and n_0 would add n_i to its routing table, then the response to a `FIND_NODE` message received after the `ACK` r_1 acknowledging m_1 is sent *must* include n_i if it would be appropriate to do so.

You do *not* need to optimize your communication or operations for concurrent access. For example, if I call `FindNode()` simultaneously for two different IDs that are near each other in the DHT, it is OK if you contact some nodes more than once (once for each `FindNode()` resolution) because they are proceeding in parallel without any knowledge of the other operation.

The `Neighbors()` method on your DHT requires multiple operations to be performed on the routing table, if you use the public routing table API. In particular, it requires that you call `Buckets()` followed by a series of `GetNodes()` operations. This is racy, and could lead to inconsistent state being returned if the routing table changes during this process. (For example, consider calling `GetNodes()` on the "last" bucket that was returned by `Buckets()`, but in the meantime `InsertNode()` has been called on the routing table and that bucket was split.) You do not need to handle this case, and my test will ensure that when I call `Neighbors()` the DHT is otherwise quiescent.

2.9 Other APIs

You are *not required* to implement the routing table APIs from Part 1 of this project for Part 2. This includes:

- The function `impl.NewRoutingTable()`
- The interface `kdht.RoutingTable`

If you have an implementation of this API from Part 1 that you wish to use, however, please do!

2.10 The Given Router

You will be provided with a `RoutingTable` implementation that replaces the `impl/routing.go` in the given code and connects to a local routing server. You must carefully install the routing server according to its instructions. Once installed, using the associated routing API will provide a complete, working router implementation.

3 Differences from Kademlia

The differences from Kademlia in this project are *all* to simplify your implementation, or to provide opportunities for partial credit. There are a number of things described in the Kademlia paper that you are not required to handle. In particular:

- You do not need to rearrange nodes in your buckets to "age" them, or remove them from your routing table when they do not respond.

- Because nodes do not age or time out, they do not need to refresh themselves.
- You do not need to keep track of potential replacements that you learn about, and then add them to your routing table only when you prove them or require them. If you find out about a node that can be added to your routing table, you should immediately `PING` it and add it when it responds. If that bucket is full, even if a node within that bucket has failed, you simply drop the new node.
- You do not need to rebalance keys or storage when nodes join the DHT. If a node joins the DHT and becomes one of the k closest nodes to a stored key, it will not learn about that key until some other node calls a `Store()` operation directly. You do not need to time out stored values.
- Because stored values do not time out, you do not need to refresh stored values.

Guiding Principle: I tried hard to take all of the needlessly fiddly parts of this project and set them aside. If something seems unusually difficult compared to the other parts of the project, ask about it. You probably don't actually have to do it. Don't take that for granted, but do get clarification!

4 Guidance

Closest k: When computing the closest k nodes to a given node during a recursive lookup, remember to consult *both* the nodes already known from previous steps in the lookup *and* the k nodes returned by a remote node! While the likelihood of the local routing table knowing a closer node than the k returned nodes becomes vanishingly unlikely as the node being sought gets farther away, for most distances it is nonzero.

Recursion: Note that while the operations on the `api/kdht.Node` interface are recursive, the network requests of the same names are *not*. For example, when a node receives a `FIND_NODE` message on the network, it should find and return the k closest nodes *in its local routing table*, while a `FindNode()` call on its interface should perform a recursive lookup for the k closest nodes *in the entire k-DHT structure.*

The three iterative operations `FindNode()`, `FindValue()`, and `Store` are performed in a similar manner, so `FindNode()` is a good example of their basic functionality. The Kademlia paper describes the `FindNode()` operation, and you should read its description carefully, but here is a sketch:

- The stopping case for `FindNode()` is: the local node has contacted the k nodes that it knows of that are closest to the ID being searched, and every one of those nodes has either *1)* not responded at all (it is failed), or *2)* not provided any node closer than the closest k that have been queried.
- The iteration process is:
  1. Create a set of the closest k nodes to the destination *from the local routing table*.
  2. Contact α of these nodes (arbitrarily, you can use whichever α of them are most convenient to you) and send them `FIND_NODES`. Merge the responses into your current set of the closest k nodes to your destination.
  3. Either stop (if the stopping case has been met) or iterate:
     (a) If there are any nodes in the closest k that have not been queried, return to 2, above.
     (b) Otherwise, return the set of closest nodes that you have built.

The `FindValue()` call is essentially similar except that it sends `FIND_VALUE` messages and that it terminates immediately if a `VALUE` message is received in response from any node. `Store()` operations are more or less a `FindNode()` followed by sending `STORE` appropriately.

Socket handling: It probably makes sense to structure your application so that *logical operations* are managed by a single goroutine (maybe the caller's goroutine) and complete on a sequential set of socket connections. Be sure to close your sockets (whether dialed or accepted) when you are done with them! For example, if `FindNode()` traverses three hops to do its job, it makes sense to create α goroutines at the first hop, each of which connects to a host and blocks to receive its reply. When the *first* such goroutine receives a reply, gather it (perhaps using a `select` statement?) and iterate, sending requests to nodes at the second hop via *alpha* goroutines as well. Once all of the sequential operations have completed, and the k closest nodes to the target node have been identified, only then does the `FindNodes()` function return to the caller. In each case, the goroutine that created a socket also read the response on that socket and then closed it when it finished.

5 Grading

This project is worth 15% of your course grade. It will be evaluated for correctness according to the requirements in Section 2, above. Note that the Autograder provided to you during implementation of this assignment will not reflect this complete grading rubric. The provided Autograder tests will ensure only that your project compiles and implements some basic subset of the required functionality. You will have to implement your own tests!

Some requirements (such as the requirement that your API implementation does not print extraneous information to standard output) do not have specific grading criteria, but failing to maintain them will cause many criteria to lose points.

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

Points	**Description**
`NewNode()` produces a valid k-DHT node	2
`Shutdown()` releases the k-DHT node's resources	1
`Ping()` and `PING` behave as described	5
`Store()`, `STORE`, and `GET` work	5
k-DHT operations on a large DHT work correctly	7

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.