Chair of Network Architectures and Services
Department of Informatics
Technical University of Munich

Peer-to-Peer Systems and Security

Team number:	52
Team members:	Roland Bernhard Reif, Manfred Stoiber
Advisor:	Richard von Seck, Filip Rezabek, Jonas Jelten
Supervisor:	Prof. Dr.-Ing. Georg Carle
Begin:	04/2021
End:	09/2021

Project Documentation

1. Our Architecture

We have separated our implementation into three main modules, namely the API communication, the peer-to-peer communication and the kBuckets. The API communication listens for newly to be established connections and listens on these connections for new orders for our distributed hash-table. These orders can be of type dhtPUT and dhtGET. As an answer, a dhtSUCCESS and dhtFAILURE message can be received.

The peer-to-peer module is responsible for parsing the messages we use for peer-to-peer communication and for the logic of how to respond to specific P2P messages.

kBuckets are an implementation specific datastructure for storing known peers). The kBuckets module handles these k-buckets e.g. populates the k-buckets and updates them.

Config

The path to a configuration file can be provided during the execution of our program with the ”-c” flag.

We have the following configuration parameters that can and should be included in the configuration file(s):

host_key	file in which private key of peer is stored
api_address	API address of peer
p2p_address	P2P address of peer
maxTTL	the maximum time to live of stored key-value pairs
k	Kademlia specific replication factor
a	Kademlia specific parallelisation factor
preConfPeer1	P2P address of another peer
preConfPeer2	P2P address of another peer
preConfPeer3	P2P address of another peer

The preconfigured peers are used during initialization so that a new peer can join the network. the API and the P2P addresses are expected to be in the format specified in the specification. A public key is not expected; we calculate the public key during the initialization phase directly.

An example of a config file can be seen in Figure 1.

```
hostkey = config/hostkey5.pem

[dht]
api_address = 127.0.0.1:3009
p2p_address = 127.0.0.1:3010
maxTTL = 86400
preConfPeer1 = 127.0.0.1:3012
preConfPeer2 = 127.0.0.1:3014
preConfPeer3 = 127.0.0.1:3016
k = 5
a = 3
```

Figure 1: Class diagram of our implementation of the API messages.

Logging

For logging aspects, we used the popular and recommended logging framework “logrus”, which is also used in projects like Docker. Every application related text output is logged by logrus. Only the outputs of the test cases are written directly to stdout using the builtin function “fmt.Println(…)” Every logged message belongs to one of following log levels: 1. Debug: Unimportant messages useful for debugging 2. Info: messages regarding program state (started, stopped, etc.) 3. Error: error messages which are not fatal 4. Panic: fatal error messages which can possibly be handled 5. Fatal: fatal error messages which cannot be handled

The desired log level can be set as the environment variable “LOG_LEVEL” with following values:

Debug	LOG_LEVEL=debug
Info	LOG_LEVEL=info
Error	LOG_LEVEL=error
Panic	LOG_LEVEL=panic
Fatal	LOG_LEVEL=fatal

Main

In our main() function, first we start logging. Second, we parse the config file. In a Goroutine we concurrently start both the APIMessageDispatcher, and the P2PMessageDispatcher. Then we initialize our peer, our local store, global parameters and other important aspects.

API Architecture

The format of the API Architecture was provided in the specification. Internally we use structs of type ”apiMessage” to represent them. Each apiMessage consists of a header, a body, and the data which is the byte representation that was received or that will be sent.

The apiHeader consists of a size and a messageType attribute. The apiBody interface is implemented by ”putBody”, ”getBody”, ”successBody”, and ”failureBody”. The interface promises the ”toString()”

method returning a string representation for logging and debugging. It also promises a ”decodeBodyFromBytes(*apiMessage)” method. This takes a pointer to an apiMessage as input. Based on the ”data” field of this apiMessage the relevant parameters for the respective apiBody is extracted, and the apiBody is built and set in the apiMessage. The apiBody interface also ensures the ”decodeBodyToBytes()” method which returns a byte representation of an apiBody that has its fields set.

A class diagram of this concept can be seen in Figure 2.

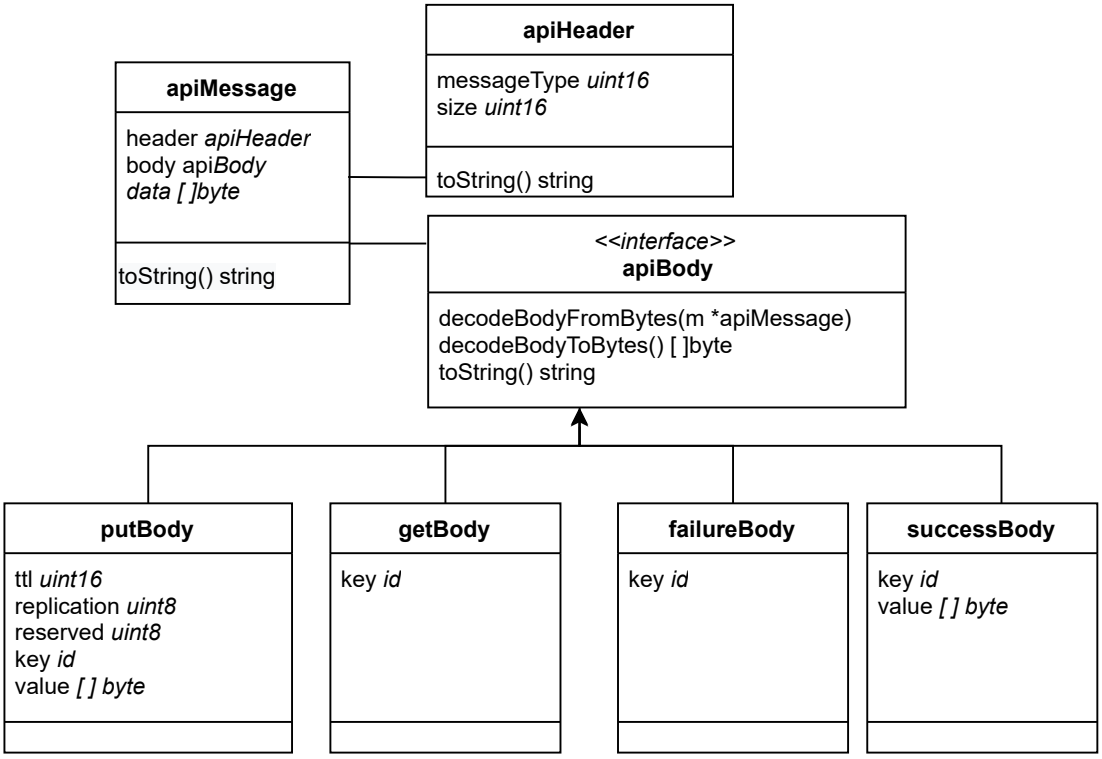

Figure 2: Class diagram of our implementation of the API messages.

The API message dispatcher listens for API calls. If it receives one, it accepts a new connection and concurrently runs the ”handleAPIconnection(net.Conn)” function. This function reads the newly received message. Then the newly received message is parsed.
If it is a dhtPUT message, then the handlePut() function, which performs the node lookup to find the k closest peers in the network. To all these peers a kdmSTORE message is sent, so that they can store the key-value pair. In addition as a caching mechanism the newly received key-value pair is stored locally. If we instead received a dhtGET message, then the handleGet() function is called. This function first searches locally for the key-value pair. If it is not found it performs a value lookup. (Note: the value lookup is also performed by the ”nodeLookup(id, bool)” function.). If the key-value pair was found a dhtSUCCESS message is sent back to the client of the API call. If not, a dhtFAILURE message is returned.

3. Peer-to-Peer Architecture

As a peer-to-peer protocol, we use our own version of a Kademlia [?] protocol as it is one of the most distributed hash table implementations.

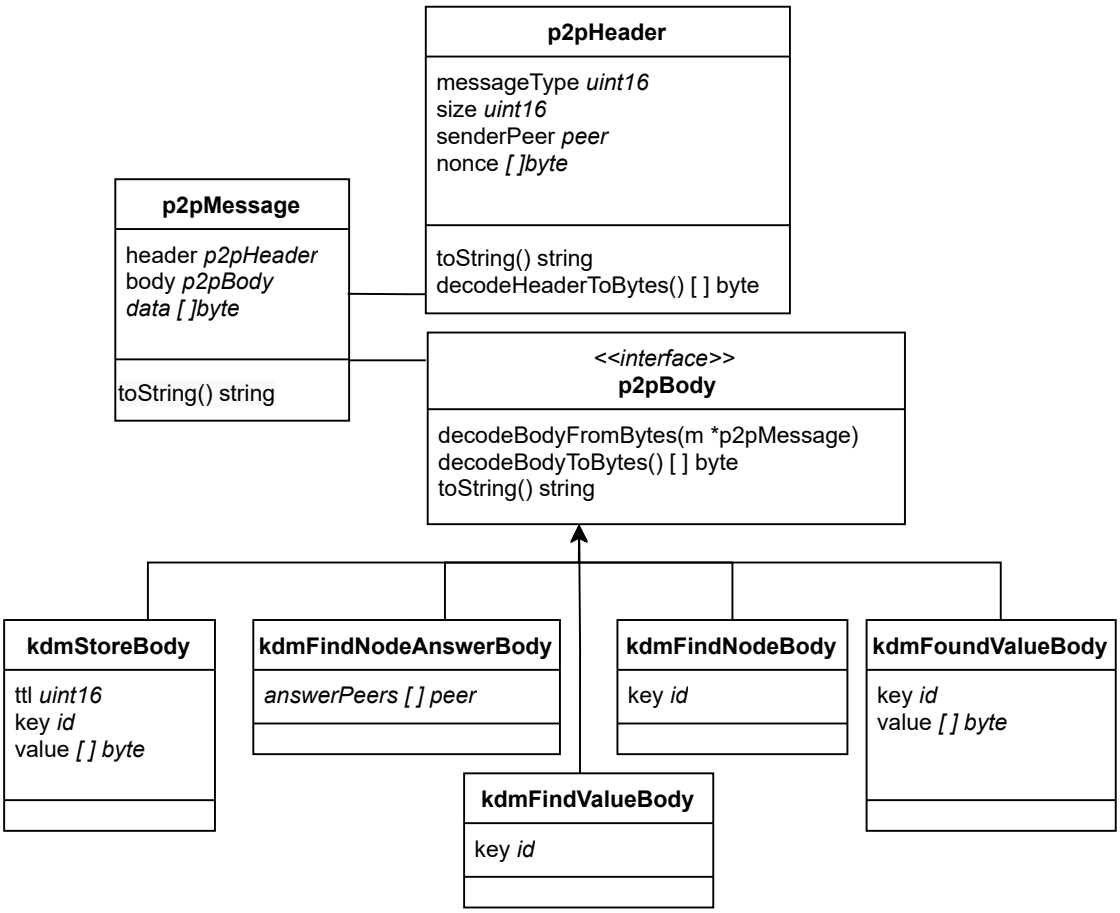

Figure 3: Class diagram of our P2P messages.

initializeP2PCommunication() First we read the private key from the file provided in the configuration file. Then we calculate the fitting RSA public key. We use this public key to generate the ID of the peer according to the specification. Next we initialize our routing tree. We also contact the specified preconfigured peers and send them a KDM_PING for initial contact of the network.

Peer vs Node For better clearness we introduced the separation between peer and node. We use the term peer for representing triples of <id, ip, port>. Node is used for representation of the local peer of our network and contains peer as well as the hash table and the routing tree. Routing table The routing table for storing known contacts is implemented as tree. Therefore we mostly call it routing tree and is implemented by the struct routingTree, which represents one node of the tree. Every node is responsible for every contact whose id starts with a given prefix. Every node contains the either the k-Bucket or the reference to its left and right children. The left child is responsible for every contact whose id starts with given prefix plus “0” and the right child for contacts with given prefix plus “1”. Additionally every node except the root node holds the reference to its own parent node.

HashTable The struct hashTable represents the local data storage of a node. The values are stored in a map which takes ids as keys and byte arrays as values. Additionally, the hashTable contains two more maps for storing the expiration times and the republishing times of each <key, value>-pair.

Expiration time Every <key, value>-pair stored in the network has a specific time to live. This is set according to the dhtPUT message or the specified maximum TTL from the configuration file. After this specific time, the <key, value>-pair should be removed completely from the network. The function "expireKeys()" is responsible for removing all <key, value>-pairs whose time to live has run out.

Republishing Due to CHURN, there are two possible problems: If all k nodes which store a <key, value>-pair leaves the network, the <key, value>-pair is lost If k nodes join the network which are closer to the key of a stored <key, value>-pair, the <key, value>-pair cannot be reached anymore To avoid those problems, we implemented key-republishing. Every node republishes every <key, value>-pair once every hour. To prevent from a huge number of messages jamming the network every hour, we optimized this naïve implementation. When a node receives the STORE-Message for a already stored <key, value>-pair, we assumes that this Message was also sent to the remaining k-1 closest nodes. Therefore, the node sets the republishing time of given <kev, value>-pair to one hour in the future. As long as the republishing times are not exactly synchronized, a <key, value>-pair will only be republished once every hour.

Node lookup The nodeLookup function is responsible for locating the k closest nodes to a given id on the network. It has a flag "findValue" which switches between finding value which is necessary for KDM_FIND_VALUE and finding nodes, which is necessary for KDM_FIND_NODE and KDM_STORE. If findValue is set, then after each round it looks if the value was cached or found and if so, it halts the lookup process and returns the found value. If the value was not found yet, it locates the k closest peers on the local node and sends KDM_FIND_VALUE or KDM_FIND_NODE messages to them, depending on the flag findValue. Afterwards it waits a specified amount of time to let contacted nodes respond and for the routing table update. The waiting time starts with 10ms and increases up to 1000ms in steps multiplied by 10. So if after at maximum 1110ms no new contact was found, the process is finished and the found contacts are returned.

Message Dispatcher The P2P Message Dispatcher listens for and accepts incoming P2P connections. Incoming connections are delegated to the "handleP2PConnection(…)" function. At first, this function initiates an update of the routing table. Afterwards it handles the incoming messages based on the message type as follows:

KDM_PING	Responds with KDM_PONG
KDM_STORE	Writes the received <key, value>-pair to local hash table
KDM_FOUND_VALUE	Writes the found <key, value>-pair to local hash table
KDM_FIND_NODE	Calls FIND_NODE(…) function to find the k closest nodes on local node and returns them to the sender
KDM_FIND_NODE_ANSWER	Updates the routing table accordingly
KDM_FIND_VALUE	Looks for value in local hash table. If found, then replies with KDM_FOUND_VALUE, else same behavior as with KDM_FIND_NODE

P2P Messages The format of the p2p messages can be seen in Figure 4

k-Bucket

A k-Bucket ist the “heart” of the Kademlia protocol and used for storing known contacts and is realized as a ordered list of peers. On each node there exist $0 < i < 256$ k-Buckets. Each k-Bucket is responsible for $2^i - 2^{(i-1)}$ contacts and is storing at maximum k of them. “k” is a system wide parameter and configured in the “config.ini” file. Due to the fact that k-Buckets with low number i are mostly empty, each node does not a priori store 256 k-Buckets, but at first only one which is responsible for all contacts and creates more buckets after time when needed. Each k-Bucket is sorted by the time the contacts are last seen. Most-recently seen are stored at the tail. This sorting is guaranteed by the update algorithm implemented in “updateRoutingTable(…)” and works as follows: Every time a node receives any message, it calls this update function Find the responsible k-Bucket for given peer If peer already exists in k-Bucket, move it to the tail. If not, check if k-Bucket is already full If k-Bucket is not full, insert peer. If it is full, check if k-Bucket is responsible for id of local node If it is responsible for id of local node, split k-Bucket and repeat insertion attempt. If it is not responsible for id of local node, ping least-recently seen node of k-Bucket. If this node responds, discard new peer and move the responding peer to the tail of the k-Bucket. If it does not respond, remove it from k-Bucket and insert the new peer

Security aspects

For ensuring availability and hence security, we store key-value pairs multiple times (usually k times) and the Kademlia protocol that we use has built in resilience against node-poisoning attacks. To prevent Denial-of-Server attacks we usually only sent data via a TCP connection, but do not wait for the answer. E.g., if we send a ”FIND_NODE” message, we directly close this connection. The ”FIND_NODE_ANSWER” message is sent via a new connection. Only for our ping-pong approach we wait for an answer. To keep long known (and hence probably trustful peers) in our kBuckets we only throw peers out of our kBuckets when they become unresponsive.

One peer in our network is able to handle 1000s of concurrent calls and connections to its API and P2P addresses.

We excluded extensive error handling to prevent malicious messages or connections from corrupting the system. Messages that do not correspond to our formats or that are contradictory (e.g. size field does not fit actual size) are dropped.

Unlike the original Kademlia paper proposed, we use a different initialization mechanism including three peers that are contacted. Hence, we can still join even if one of those peers is not reachable.

We thought about calculating the ID of a node also based on its IP and port so that malicious nodes can not directly influence their positioning. We stepped back from this idea due to the specification (e.g. the third bullet point in chapter 1).

2. Software Documentation

Running the Software: Run Executables

We have included two executables (DHT-16 and DHT-16.exe for Linux and Windows system. Those can be run directly without the need to install further dependencies. On Linux systems execute this on the terminal:

```
chmod +x DHT-16
./DHT-16
```

On Windows systems execute this on a PowerShell or on the commandline:

```
.\DHT-16
```

Running the Software: Compiling and Running Go Files

Alternatively, you can compile and run the go files directly. For this you need to have Go installed on your system. It might also be necessary to first install two dependencies. Execute the following commands:

```
go get gopkg.in/ini.v1
go get github.com/sirupsen/logrus
run .
```

Useful Tools for Running Multiple Peers

Create Configs and Hostkeys

We have written two bash scripts that are useful for testing. With config/createConfigsAndHostKeys.sh you can create multiple config files and private hostkeys. It might be necessary to make the file executable with "chmod +x config/createConfigsAndHostKeys.sh". The "n" variable can be used to specify how many config files (and private host keys) you want to create.

Run Peers Concurrently

The second bash script (runPeersConcurrent.sh) starts up a number of peers concurrently. Again it might be necessary to make the file executable first. Again you can set the number of peers to be started with the variable "n".

Run our Go Tests

You can run all our Go tests with the "go test" command. Our test can be run both with multiple peers running and also with only one peer running. In this case you will see the logging of a few error messages (because this one peer can not reach the three pre-configured peers), but all tests still pass. We have an average test coverage of 81% of our lines. A more detailed overview of test coverage can be seen in Figure 5.

Additional to testing, we used static code analysis for further quality assertions. We used the "linter runner" and followed all useful recommendations.

Known issues

We use two different methodologies for answering a KDM_PING message with a KDM_PONG message. During nodelookup we wait on the same connection for the pong message where the ping was sent. During initialization we send the pong back on a new connection.

3. Future Work

Further security messages can be introduced. Currently the implemented nonce that we send with our DHT messages are not really used. But for future work they can be usefully. E.g., peers can use them to prevent replay attacks. Signatures can be added to ensure the integrity of sent messages. Assuming that peers have exchanged the public keys prior to the start of our P2P system, asymmetric encryption can be added to ensure confidentiality.

4.-5. Workload Distribution and Effort Spent

Manfred Stoiber

Manfred has implemented the tree structure of the routing table and the k-Buckets as well as the algorithms for updating and modifying them. He also implemented the functions needed for the management of the contacts in k-Buckets, for example the algorithms to split buckets, find responsible k-Buckets, find number of closest peers etc. Besides that, he implemented the hash table and the expiration and republishing mechanism. He complemented the node lookup and wrote the test cases in kBuckets_test.go and P2PCommunication_test.go.
His estimated overall time spent for the project is about 140 hours and since it was his first project with Golang ever, the learning (and sometimes frustration) process took about 30h additionally.

Roland Reif

Roland has implemented all API and P2P messages, their structure, their respective coding and decoding, and the corresponding explicit functions to send and receive those messages. He also implemented most the Main function, the parsing of the configuration file and the initialization of peers (e.g. with ”initializeP2Pcommunication()”). Additionally he coded the basic structure of the node lookup and some helper functions (e.g. ”wasAnyNewPeerAdded()”). He implemented the tests in APIMessages_test, APICommunication_test, and P2PMessages_test. He also wrote the bash scripts.
His estimated overall time spent for the project is about 170 hours.

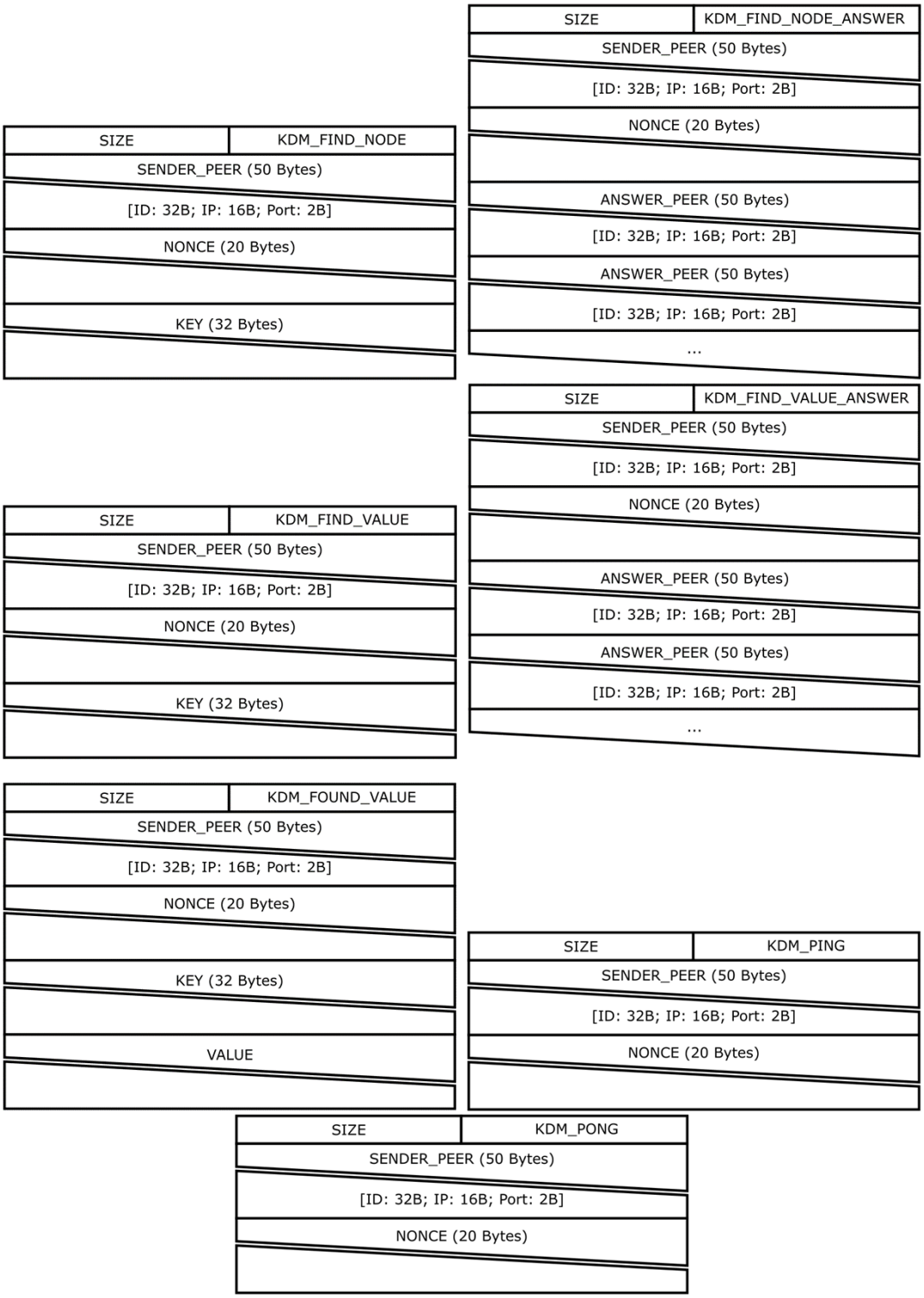

Figure 4: The format of the P2P messages.

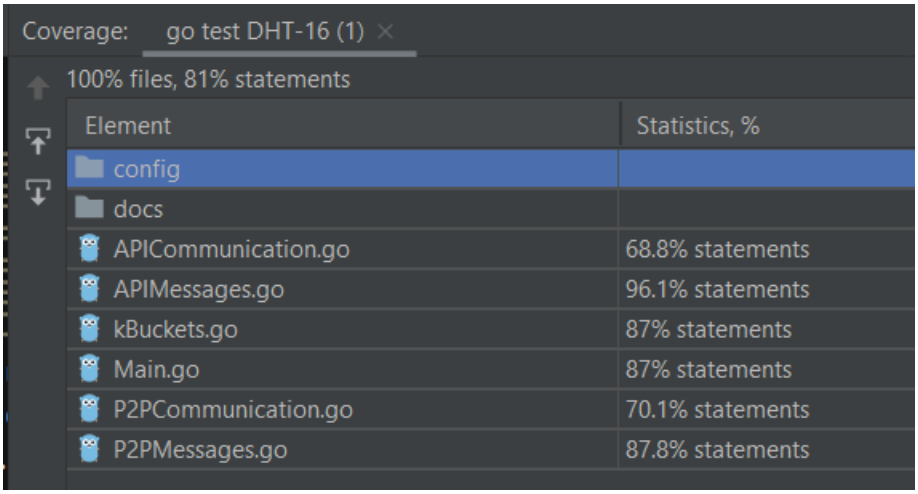

Figure 5: Test coverage.