Peer-to-Peer Systems and Security

Midterm Report

Team 17
Ege Oztas
ge85jon@mytum.de
Loris Nathan Verga
go72jak@mytum.de

Contents

1 Changes to our assumptions in the initial report

Since our progress thought the midterm phase of the project not much has changed in our assumptions, scope and methods of the project.

We did however saw that in our initial positioning we misunderstood the message types that we were allowed. We thought that we were to use only the given 4 general DHT message types, but on a re reading of the specification we saw that we were allowed to create our own messages in a given cut of the

message palette. This revelation has given us more breathing room in the implementation of a Kademlia model which in its own specification needed specific message types.

We also realized that the other modules would use our module using the DHT PUT and DHT GET message interface. We therefore had to think up a separation between the messages used for the peers involved in creating the DHT and the messages used to communicate with the other modules.

While also working with the networking functionality we have gone through multiple iterations of network package handlers where we first tried raw socket networking and multi threading then after observing the given examples by the teaching staff switched to using asyncho library. We also discovered at a late stage that the use of Future construct could help us implement certain functionalities.

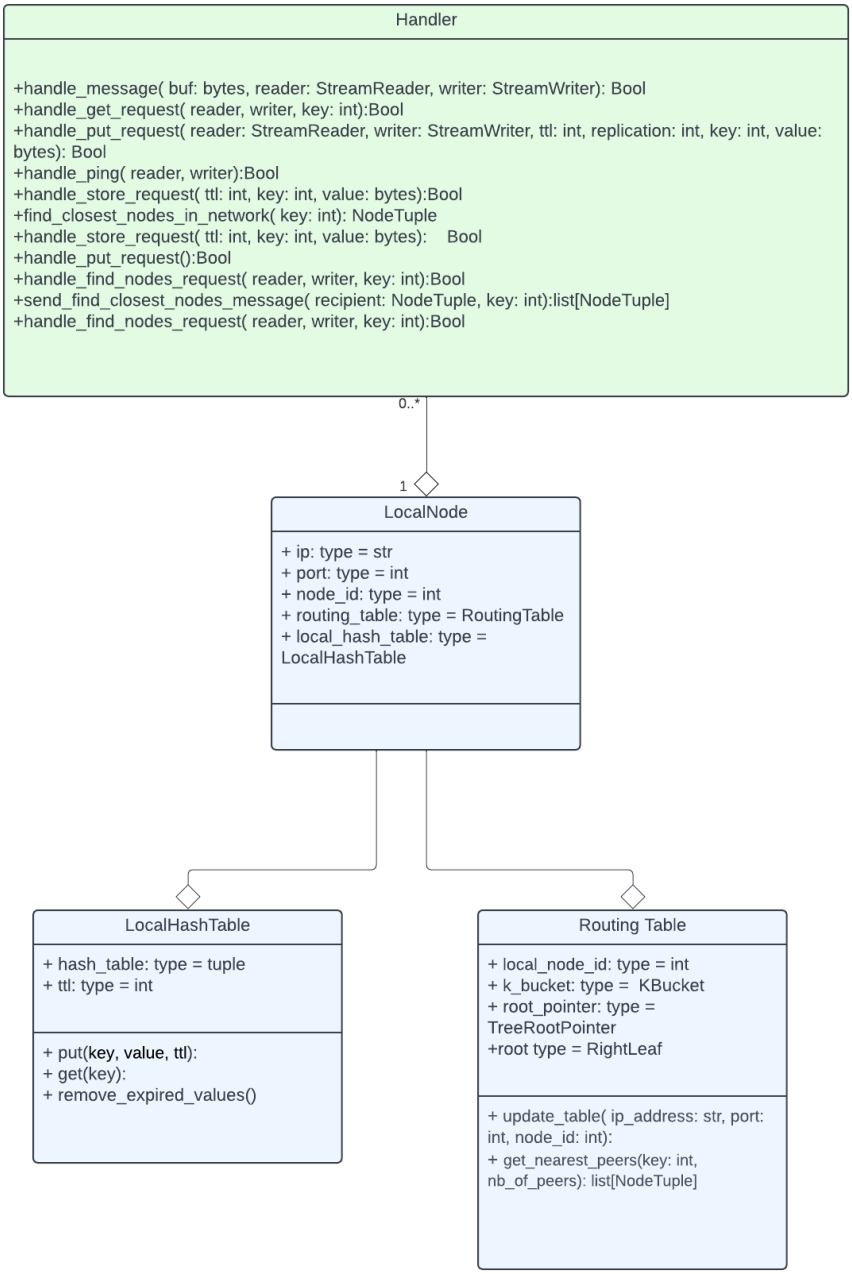
2 Architecture of the module

2.1 Request handler

Here on the class diagram shown on the figure 1 you can observe our DHT agent's class diagram. Handler is the class that defines the network inferface of our DHT module. It is used to communicates with the other peers participating in the construction of the DHT and for the communication with the other modules of the VoidPhone project. The class contains a handle message function that takes an incoming message as a parameter and then calls a specific function responsible for handling the specific request. The class therefore also naturally contains all the specific message handlers associated with all the specific requests.

The class has a single local node representing the stored local information that is in possession of the peer. LocalNode itself contains a hash table that serves as the local storage and a RoutingTable used for the global routing. Local node also has its own id that it shares with other peers.

Figure 1: Class diagram of the Handle class.



2.2 DHT Client

As mentioned previously, the Handler class manages messages originating from both other peers participating in the creation of the DHT and from other modules of the Voidphone project. Testing the use of our DHT would therefore involve using another module to send requests to our module. To facilitate interaction with our module without having to use others, we plan to create a DHT client that will allow data to be stored in the DHT using a simple interface. This client will run in a separate process and communicate with the DHT module asynchronously via network messages. The client has not been implemented yet, but we are planning to do so.

2.3 Usage of the Kademlia Paper

For this project, we have decided to create a implementation that follows as close as possible the specification given in the Kademlia paper [1].

2.3.1 Routing Table

As specified in the Kademlia paper, the routing table is used to locate peers that possess a specific piece of data or to determine which peers should store a specific piece of data. Each peer has a local routing table, which is a tree data structure containing K-Buckets that, in turn, hold information about other peers.

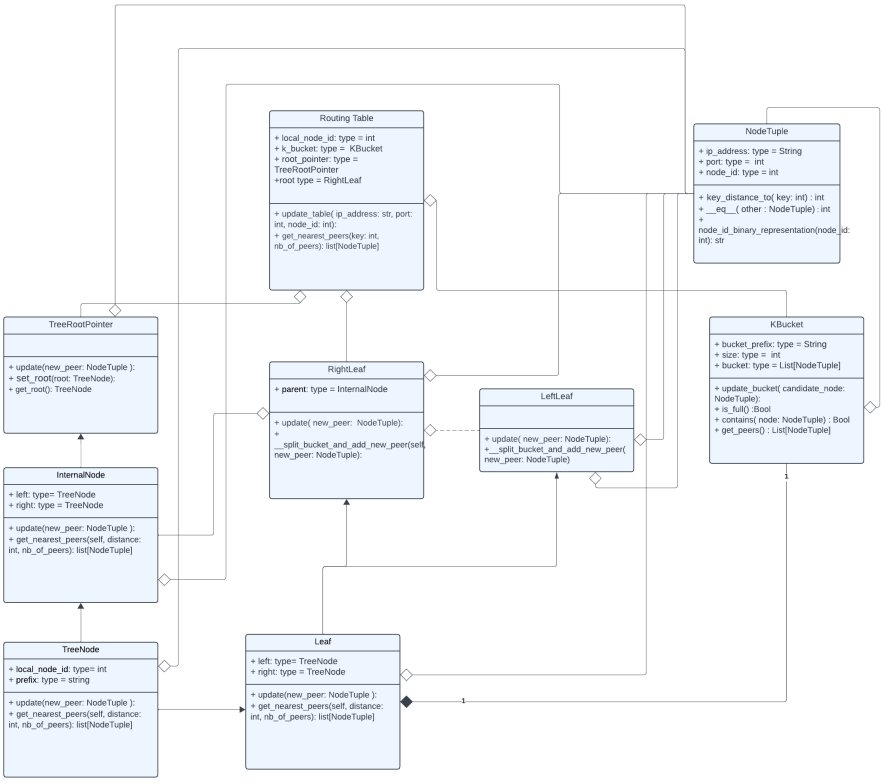
A K-Bucket has a fixed size of k elements and contains information about peers located at a range of distance from the local node. Here, the distance is evaluated using the XOR metric, and the distance is measured between the two node IDs. A K-bucket covers a prefix of the binary representation of the distance to the ID of the local node. The K-buckets are organized in the tree of the routing table based on the distance prefix they represent. For example, the root has an empty prefix (""). Then, the subtree defined by the right child of the root will have a prefix of "0". Only the leaves of the tree contain K-buckets.

We have chosen to use a recursively allocated tree data structure for the routing table. One major advantage is that the local node allocates only the minimal number of K-buckets needed to store information about the nodes it knows. For example, if it knows fewer than K nodes, the tree data structure will consist solely of a root with a single K-bucket. Without this dynamic allocation, the node would have allocated 256 K-buckets for fewer than K nodes.

The Kademlia specification defines that the routing table stores a maximum of k nodes per power of 2 distance value. This means that the routing table knows fewer nodes as the distance to these nodes increases. This is why we have defined two types of leaves in our tree structure. A right leaf contains a K-bucket with a prefix consisting solely of zeros. This bucket can be split by allocating two new buckets that define more specific prefixes. The left leaves have a prefix containing a 1, and their bucket cannot be split because they represent distant buckets.

The class diagram is presented in Figure 2.

Figure 2: Class diagram of the Kademlia's RoutingTable class.



2.3.2 K-Buckets

As previously mentioned, K-buckets contain information about peers whose IDs match the prefix of the K-bucket. If the K-bucket is full and a new peer is to be added, the K-bucket may either split or discard a peer based on the bucket's prefix. The K-buckets implement a least-recently seen eviction policy but the nodes always online are never removed from the list. The nodes information stored is constituted of the IP, the port and the node ID.

3 Security Measures

3.1 Data availability ensured by redundancy across peers

During a store operation, the DHT module locates the k closest nodes to the key of the data to be stored. These nodes are then responsible for storing the data. By having this redundancy, we eliminate the single point of failure that could lead to data loss. Additionally, thanks to Kademlia's key-republishing mechanism, key-value pairs are periodically republished throughout the network to address the issue of nodes leaving the network.

3.2 DoS resistance provided by K-Buckets

K-buckets offer a benefit in providing resistance to certain DoS attacks. Flooding the system with new nodes cannot flush a node's routing state. Kademlia nodes only insert new nodes into their k-buckets as old nodes leave the system.

3.3 Implementation reliability

For making sure of the the code and the implementation secure against the oversight on our part we have started writing unit tests for the completed parts of the code. We also wrote the our logic with exceptions for an inappropriate input or messages from peers wherever. We acknowledge that there is still work to do to make our implementation stronger against potential peer attackers.

4 Specification of the peer-to-peer protocol that will be implemented

As we have explained in the architecture of the project we implemented our DHT using the Kademlia specification.

For peers to communicate, Kademlia defines 5 additional types of message

1. **PING:** PING message is used to see if a given peer node is online or not.
2. **STORE:** This message also carries a key, value pair with it and isntructs a node to store the given value for later use.
3. **FIND_VALUE:** Takes a node id and sends it to a peer node. If the receiver has the node in the KBucket it has, it returns with the IP address, port, NodeID of the given node.
4. **FIND_NODE:** Similar to FIND_VALUE takes a node id and recipent of the message instead of a single Node tuple returns the k-nodes it knows that are closest to the given node id.
5. **DHT_FIND_NODE_RESP:** The reponse to a FIND_NODE query.

The messages defined in the specifications of the VoidPhone project for the DHT module complete this message set.

The messages DHT_FIND_NODES and DHT_FIND_VALUE are utilized by the node lookup mechanism of Kademlia. The procedure starts by the peer nodes stored in the local routing table queries them for the node id. Then after this step the same queries is performed on the new peers the local node has learned about. All of the messages of Kademlia are working with this lookup function to recursively reach to the given node.

5 Future Work

The implementation of the algorithmic part of Kademlia and most of the message handlers are completed. We still have the following steps to complete:

1. Finish the message handler and complete the unit tests for this class.
2. Add the functionality to refresh the k-buckets.
3. Create the client and add the functionality for a node to join a network with at least one known contact.
4. Test the system thoroughly with multiple instances of nodes.
5. Add in-network caching for optimization
6. Write the documentation and create the scripts for the installation.

6 Workload Distribution

Task	**Person**
Initial Network Handling	Ege
Kademlia Routing Table and K-Buckets	Loris
Network overhaul	Loris
Node lookups	Ege

7 Effort spent for the project

Up until this part of the project Ege Öztaş worked on Network handling part while Loris Nathan Verga worked with the Routing table and the algorithmic part of Kademlia. The overhaul on network handling and creation of the handling on later stages was done by Loris. The midterm report was written mostly by Ege. On deciding which way to implement kademlia they both worked together. There were weeks where one or both of us were unable to exert effort for this particular project but nevertheless we feel we have shown equal effort up until this point.

References

[1] Petar Maymounkov and David Mazières (2002) *Kademlia: A Peer-to-peer Information System Based on the XOR Metric*, New York University. Available at `http://kademlia.scs.cs.nyu.edu`.