Final report

Requirements

Java/Kotlin (JVM) Runtime supporting at least JVM bytecode 1.8 is required to run my software. There are no particular hardware requirements.

Building

To build executable jar, first download the project code:
`git clone git@gitlab.lrz.de:p2psec-2018-projects/51.git DHT`
Or
`git clone git@github.com:sssemil/DHT_TUM.git DHT`

Then cd to the cloned directory:

```
cd DHT
```

After that execute following to build the jar:

```
gradle clean build
```

Resulting jar should be at *DHT/build/libs/DHT-0.0.1.jar*.

Installing and Running

To install it, just save the jar file in a convenient place on your system.
To run this software on a Linux system execute following:
`java -jar /path/to/my/DHT-0.0.1.jar`

TODO: I am planning to make it more like a service (daemon), and alwo create a cli executable.

Simple Shell

After starting the application, you will see a simple command line interface. If you type help, you will see available functions:

```
ping: 'ping {destination IP} {destination port}'
   Check if there is an open port with our software.
list: 'list'
   List peers.
addPeer: 'addPeer {destination IP} {destination port} [optional]{remote peer ID}'
   Add new peer.
put: 'put {value}'
   Add new item. Key will be printed after executing.
get: 'get {key}'
   Get item by key.
port: 'port'
   Print current port number.
port: 'port {new port [1024-65535]}'
```

```
   Change port number. Will restart the server.
exit: 'exit'
   Exit the program.
```

Port is generated automatically by ServerSocket.

TODO: I am planning to make a separate simple admin interface over TCP.

TODO: Adding new peers is not checking a lot. It would be easy to spam it right now with malicious peers that respond to pings.

Known Issues

I had to set tolerance level to max, because in small networks the probability of finding proper peer is too low. Because of that search performance is not as good as in a normal Kademlia network.

There are not many checks for validity of incoming messages, which might cause crashes.

Storage is unlimited, so you can spam it as well, and crash the application, and overload the system.

Protocols

To communicate with the application, I implemented standart messages from specification.

DHT_PUT = 650
DHT_GET = 651
DHT_SUCCESS = 652
DHT_FAILURE = 653

KEY_LENGTH = 128 / 8 *// bytes (md5 size)*

DHT_PUT(**val size**: Short, **val code**: Short = *DHT_PUT*, **val ttl**: Short, **val replicationsLeft**: Byte, **val key**: ByteArray[*KEY_LENGTH*], **val value**: ByteArray)
DHT_GET(**val size**: Short, **val code**: Short = *DHT_GET*, **val key**: ByteArray[*KEY_LENGTH*])
DHT_SUCCESS(**val size**: Short, **val code**: Short = *DHT_SUCCESS*, **val key**: ByteArray[*KEY_LENGTH*], **val value**: ByteArray)
DHT_FAILURE(**val size**: Short, **val code**: Short = *DHT_FAILURE*, **val key**: ByteArray)

And also a object sending message format.

DHT_OBJ = 654

DHT_OBJ(**val size**: Short, **val code**: Short = *DHT_OBJ*, **val objCode**: Int, **val obj**: ByteArray)

Where **obj** is a JSON or encrypted with AES + RSA JSON of the object. There are predefined object codes, to properly parse the **obj** (pick proper class, use or not use encryption).

OBJ_PING = 1
OBJ_PONG = 2
OBJ_PUT = 3
OBJ_FIND_VALUE = 4
OBJ_FOUND_VALUE = 5
OBJ_FOUND_PEERS = 6

All objects used in *DHT_OBJ* have to extend TokenModel.

TokenModel(**var token**: Double = Math.random())

Here we have **token**, which is used to properly find replies to multiple requests.

PING(**val senderPeerId**: ByteArray, **val senderPort**: Int) : TokenModel()
PONG(**val tokenReply**: Double, **val peerId**: ByteArray, **val port**: Int) : TokenModel(**token** = **tokenReply**)

Here, in PONG, we manually set token to the one we got from PING, so it would be possible to find proper replies in case of multiple simultaneous PINGs.

PUT(**val ttl**: Long, **var replicationsLeft**: Byte, **val value**: ByteArray) : TokenModel()

PUT is used internally to ask other peers to save a value.

FIND_VALUE(**val publicKey**: ByteArray, **val key**: ByteArray[*KEY_LENGTH*]) : TokenModel()
FOUND_VALUE(**var tokenReply**: Double, **val value**: ByteArray) : TokenModel(**token** = **tokenReply**)

Peer(**val id**: ByteArray, **val ip**: InetAddress, **val port**: Int)
FOUND_PEERS(**val tokenReply**: Double, **val peers**: Array<Peer>) : TokenModel(**token** = **tokenReply**)

FIND_VALUE and FOUND_VALUE are used to lookup key and get value or closest peers (FOUND_PEERS), if value not found.

TODO: It is still possible to see the type of the object. I plan to encrypt it as well.

Peer ID generation

We use RSA to generate 4096 bit long key. This is used later in DHT_OBJ to encrypt objects.

Distance

To calculate XOR distance between two peers, first we generate md5 sum of their public keys, which is 128 bits long, and then calculate XOR from them. After that, distance is calculated with $\log_2$ of their XOR.

Keys for values are also md5 sums of the value. Distance is calculated in the same manner.

For each key bit there are buckets that can fit 20 peers. Each bucket is a sorted list, that automatically removes “worst” peer when we add one too many.

Janitor

Janitor is background task running every hour. Right now it is removing outdated items in storage, and sending all locally stored values to known peers.

TODO: Regularly check stored peers for availability. If not there, mark it as less favourable.

Efforts

I was working alone.