# P2P Systeme und Sicherheit (IN2194): Final Report

Mohamed Elzarei, Hady Mohamed

September 13, 2020

## 1 Architecture

We designed and implemented our Distributed Hash Table (DHT) based on *Chord* protocol [1]. We use *SHA-256* as our hash function to map keys on the ring. Our module is divided into two sub-modules API Module that interacts with the end-user and Chord (DHT) module that's part of the DHT Network. We use Python's AsyncIO and Event loop to run both modules with non-blocking IO in a single Thread.

1. API Module: Listens for incoming messages from clients (e.g DHT GET & DHT PUT messages) and calls the P2P module.
2. P2P Module: Responds to the API module calls and implements the DHT protocol. Communication between the nodes in the DHT module uses RPC calls.

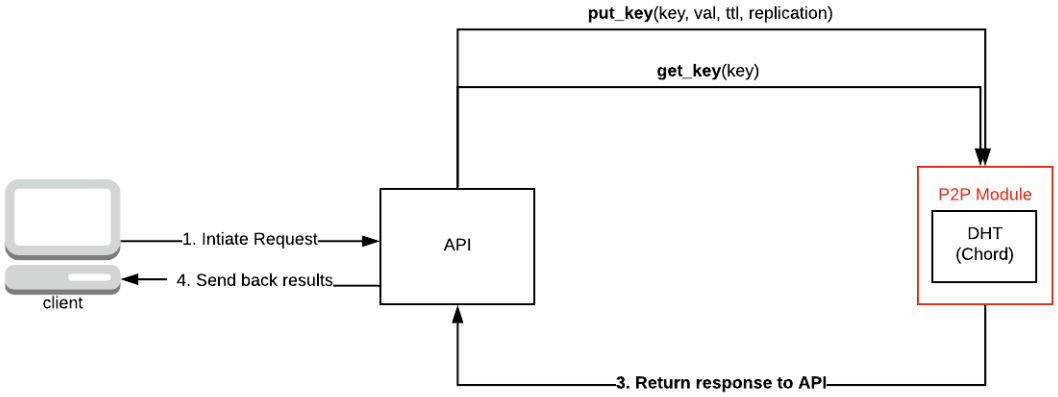


Figure 1: Request Flow

### 1.1 Communication

We use Python as Programming Language and AsyncIO to provide a non-blocking single threaded communication. We rely on DHT messages described in Section 3 for communication between the client and API Server. For communication inside the DHT Network we rely on RPC Calls based on JSON-RPC between the nodes, that proved to be much easier and less messy as we continued the implementation.

## 1.2 Security

### 1.2.1 TLS

All communications between nodes are secured using *TLS* protocol. TLS protocol aims to provide privacy and data integrity between for all the communications in our DHT Network. For a Node to join our network it must be first have it's certificate signed and verified by the CA thus no arbitrarily malicious nodes can join the DHT network. We provide a script in `src/tls` to automate the generation of new certificates for new nodes.

### 1.2.2 HMAC

HMAC is used to create digests from the messages. This provides even more security features, such as integrity and authenticity of a node's action, since each node uses its own secret key to create the digests.

## 1.3 Exception Handling

Our DHT implementation can handle failures and churn of nodes by maintaining a successors list for each node. That's if one node fails it can find the next node in the successor list to continue working correctly. We also implement the `stabilize, fix_fingers, check_predecessor` methods from the paper to periodically fix and update the node.

# 2 Software Documentation

## 2.1 Getting Started

### 2.1.1 Dependencies

The Libraries used in this project are listed in `pyproject.toml`. We tested running the project on Mac Os 10.14 and Ubuntu 18.04 (with/without Docker)

### 2.1.2 Docker

We recommend having Docker and Docker-compose [1]installed. You can then run `docker-compose up` to start a Chord Network with Single API Server and 4 Nodes.

You can edit the provided compose file to add more nodes or use the provided `Dockerfile` to customize the image.

### 2.1.3 Without Docker

Make sure to have `Python 3.8+` and `poetry` [2] installed. To install dependencies run `poetry install`. Afte that you can use python to run the main file `python3 src/main.py` options are shown in the listing below. The node can also be configured by modifying the config file in `src/config/bootstrap.ini`

```
usage: main.py [-h] [--start-api] [--dht-address DHT_ADDRESS] [--api-address
               [--bootstrap-node BOOTSTRAP_NODE]

optional arguments:
  -h, --help            show this help message and exit
```

[1]https://docs.docker.com/compose/
[2]https://python-poetry.org/

```
--start-api             If not present won't start an API server.
--dht-address DHT_ADDRESS
                        Address to run the DHT Node on
--api-address API_ADDRESS
                        Address to run the DHT Node on
--bootstrap-node BOOTSTRAP_NODE
                        Start a new Chord Ring if argument no present
```

### 2.2 Tests

We implemented tests for the basic functionality of the API and Chord to run the tests using `pytest` `PYTHONPATH=src pytest -s test/`

### 2.3 Modules Documentation

Each module, class and method in our project is documented and can be checked in code. We also used `sphinx` to generate static documentation by running the following in the project directory:

```
cd documentation && sphinx-build . _build && make html
```

We also have a live deployed version here: `https://tum-p2p-dht.netlify.app`

## 3 Protocol

With the architecture above in mind, we proceed with how we handle the P2P protocol and the different messages with their details. We decided to not use the gossiping model in our implementation, but rather rely on directly bootstrapping the first node. Each node that wants to join that DHT can reference this node to join the network or any other node in the network.

### 3.1 DHT PUT

The **DHT PUT** stores the value under the given key.

1. **Size**: The size of the DHT PUT (the payload / body) excluding any headers.
2. **Type**: The type of the message (**DHT PUT** in this case).
3. **key**: The key under which the value should be attempted to be stored.
4. **value**: The value/data being stored to be stored.
5. **TTL**: Time in seconds the key-value pair should be stored in the network. (time to live) (best effort).
6. **Reserved**: reserved for use when needed.
7. **replication**: An estimate of how many replicas of (either under different keys or different nodes), the key-value pair should be stored (also best effort).

### 3.2 DHT GET

The **DHT GET** A request to request the value stored under the provided key.

1. **size**: The size of the payload (**DHT GET**)
2. **Type**: The type of the message (**DHT GET**).
3. **key**: The key for which the value should be attempted to be retrieved.

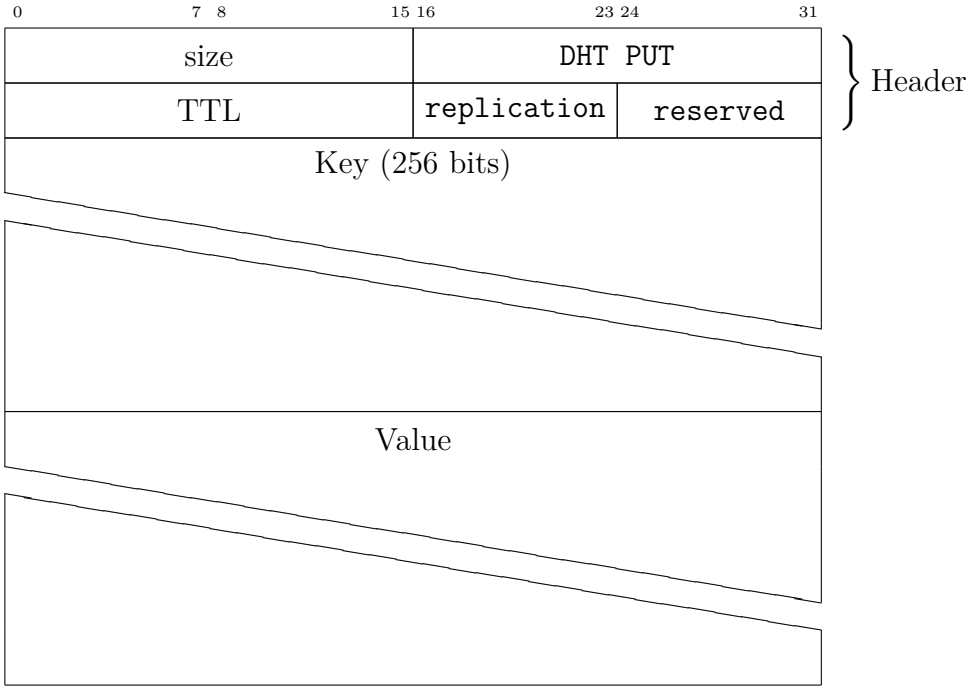


Figure 2: P2P PUT message

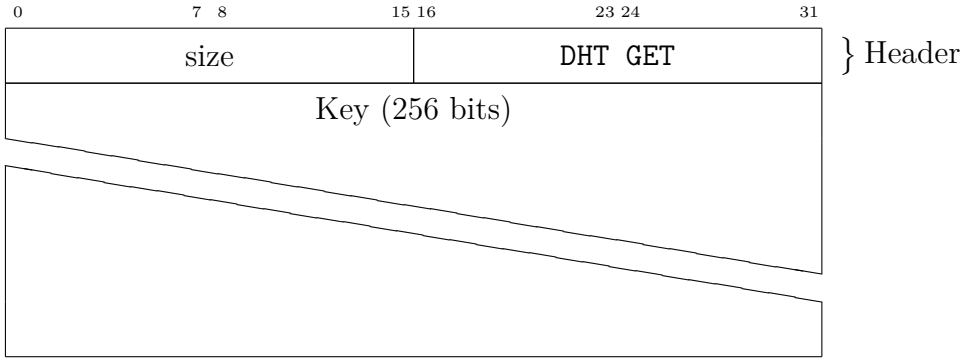


Figure 3: DHT GET message

## 3.3 DHT SUCCESS

A DHT message, which contains a **DHT SUCCESS** status, the **key** and the **value**, after a successful retrieval. (following a **DHT GET** request)

1. **size**: The size of the payload (**DHT SUCCESS**)
2. **Type**: The type of the message (**DHT SUCCESS**).
3. **key**: The requested key which maps to the value / data stored.
4. **value**: The value / data stored stored under the requested key.

## 3.4 DHT FAILURE

A DHT message, which contains a **DHT FAILURE** status, after a failed retrieval.

1. **size**: The size of the payload (**DHT FAILURE**)
2. **Type**: The type of the message (**DHT FAILURE**).
3. **key**: The requested key for which the retrieval failed.

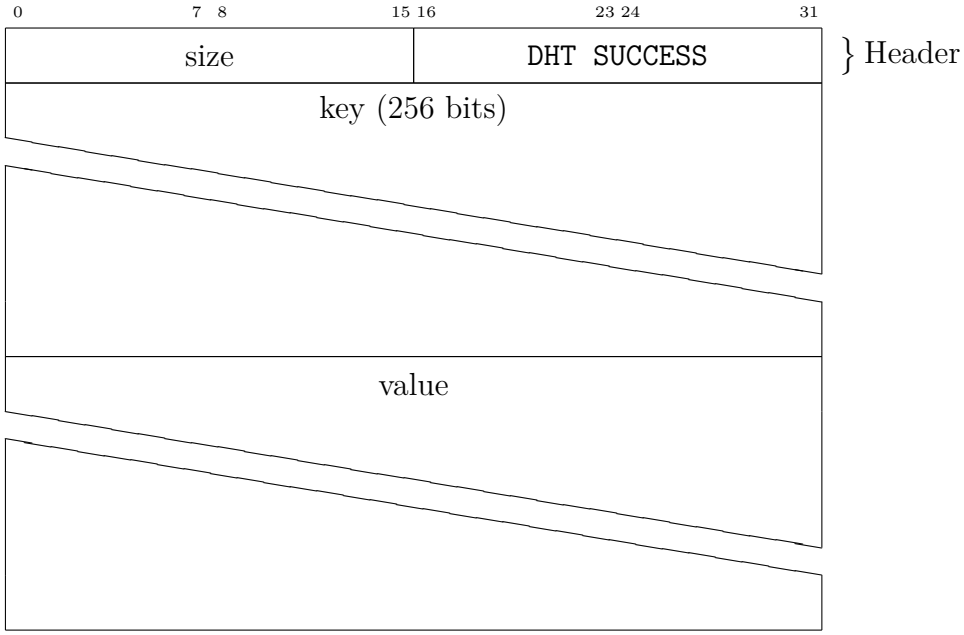


Figure 4: DHT Success message

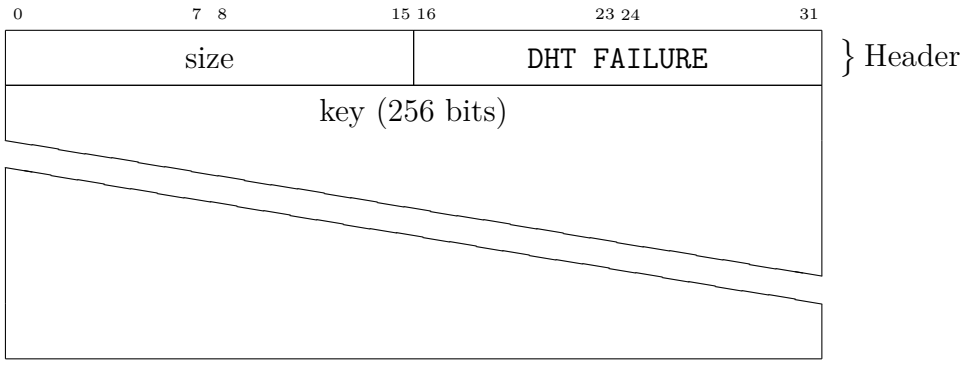


Figure 5: P2P Failure message

## 3.5 RPC Calls

1. **rpc_ask_for_succ**(node, id): Ask **node** to find the successor of **id**.
2. **rpc_ask_for_pred_and_succlist**(node): Ask **node** for its predecessor and successors list.
3. **rpc_ping**(node): Checks if **node** is still alive.
4. **rpc_notify**(next_node, prev_node): Notifies **next_node** that **prev_node** thinks it is their predecessor.
5. **rpc_get_key**(next_node, key, ttl, is_replica): Asks **next_node** to find the value associated with **key**.
6. **rpc_save_key**(next_node, key, value, ttl): Asks **node** to saves **key** it its storage.
7. **rpc_put_key**(next_node, key, value): Ask **next_node** to put **key** and its **value** in the storage of its responsible node.
8. **rpc_get_all_keys**(next_node, node): Gets all the keys the node from **next_node** that **node** might be responsible for.

# 4 Workload Distribution

We both (Mohamed & Hady) sat together to brainstorm and create the protocol and to implement the initial API server as well as project structure and create this documentation report. We both also worked on implementing the chord network as well as everything else together with the exception of **tests**, which were written by (Mohamed) and **Doc string**, which were added by (Hady).

# 5 Future Work

- Implement custom field for protocol version to uniquely identify our protocol.
- Currently,the response for RPCs (Remote procedure calls) propagates back in the ring. This can be optimized by having the last node handle this instead of propagating it back through the ring.

# 6 Effort Spent

The workload after the midterm report has been around 80% for coding and documenting, as well as structuring the project. The remaining 20% was spent reading and understanding the different parts (e.g the chord paper). Similar to last time We cant precisely bound the amount of time spent working on it, as we got carried away with the different aspects of the project.

# References

[1] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup protocol for internet applications," *IEEE/ACM Trans. Netw.*, vol. 11, p. 17–32, Feb. 2003.