Peer-to-Peer Systems and Security (IN2194)
Final Project Report

Doğan Can Hasanoğlu, Emin Sadikhov

1. Architecture

We have developed a Distributed Hash Table (DHT) based on the Kademlia protocol. The system's architecture consists of two primary components: the API Handler and the DHT Service. The API Handler establishes connections, initially with the bootstrap node, followed by additional peers. It processes incoming requests and routes them to the DHT Service, while also monitoring node liveness and adjusting connections based on specific operations. The DHT Service handles operations such as PUT, GET, FIND NODE, and FIND VALUE, ensuring efficient data storage and retrieval across the distributed network.

1.1. Logical Structure

API Handler: This component launches the server, initializes the DHT Service and node, and enables interaction within the DHT network.

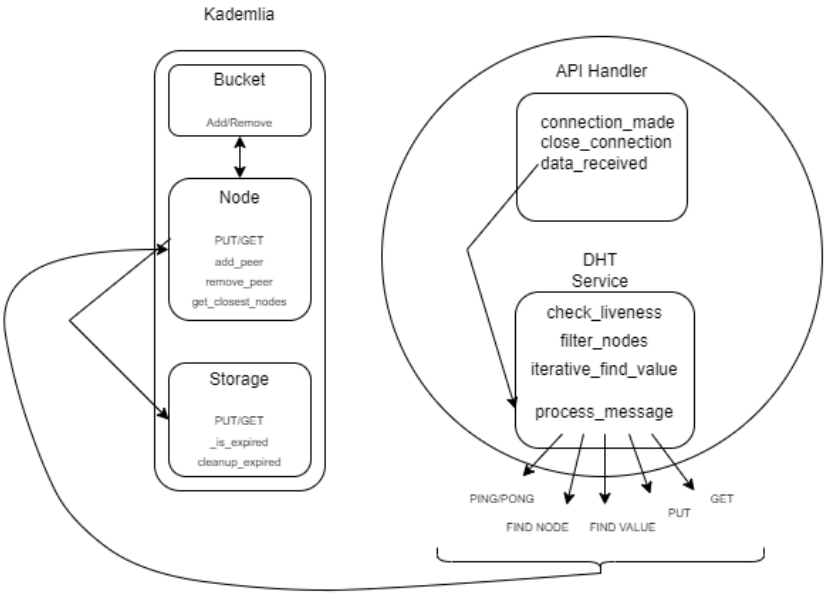
DHT Service: It communicates consistently with the API Handler, processing DHT-related incoming messages. Utilizing the Kademlia Service, it ensures operations align with the protocol's standards for various operations and message types.

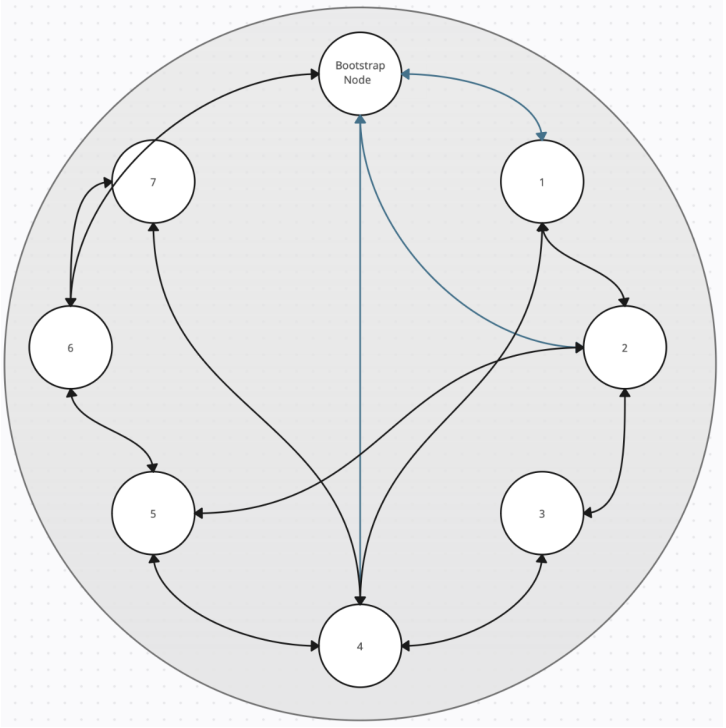
1.2. Process Architecture & Networking

We've implemented the Kademlia protocol according to our initial architecture. Using Asyncio, our node can handle multiple client connections for PUT and GET operations. Basic exception handling is included to manage various scenarios and maintain functionality.

1.3. Security Measures

We used SHA-256 for hashing operations and turned data into a 160-bit fixed-length string of characters. This modification works well with our Kademlia system for finding and storing data. To make the data transfer safer, we use SSL certificates. We also set up important security keys in advance and use a Certificate Authority for extra safety. Integrity checks are consistently conducted, providing an additional layer of assurance regarding our data's security and reliability.





2. Software Documentation

The Libraries used in this project are listed in pyproject.toml. We tested running the project on Mac Os 13.2.1 Ventura, Ubuntu 22.04 and Windows 11 Home Edition.

2.1. Configuration and Build

Make sure to have Python 3.11 and poetry installed. To build the project and install dependencies you can run the below commands:

tar -xzvf DHT-8-main.tar.gz (assuming downloaded tar.gz file)

pip3 install poetry or ***curl -sSL https://install.python-poetry.org/ | python3***
Add Poetry to PATH (***export PATH="/path/to/dir:$PATH"***)
Make sure environment runs on ***Python 3.11***
poetry install (Install dependencies)
poetry shell (Enter to virtual environment every time you try to run a node)

usage: `python3 main.py [-a IP address][-p port_number] [--bootstrap]`

optional parameters:
-a : If not specified, retrieves the IP address from the config file (default:localhost).
-p : If not specified, retrieves the port from the config file (default:6501 - listen port and 7501 API port).

Note that --bootstrap parameters are not necessary when starting the bootstrap node.

The API address is by default assigned to listen_address + 1000. (eg. 6501:7501)

Example:

`python3 main.py -a 127.0.0.1 -p 6501` (Starts bootstrap node)
`python3 main.py -a 127.0.0.1 -p 6502 --bootstrap` (Starts peer and connects to bootstrap node)

2.2. Tests

Tests can be run using two methods:

1. The `test/` folder contains pytest tests. To execute all these tests, navigate to the root directory (`DHT-8/`) and run `***pytest test***`.

2. The `***src/voidphone_pytest_dht***` folder contains various other tests. For example, to run `***dht_client_test.py***`, use the following command:

usage: `python3 dht_client_test.py -a 127.0.0.1 -p 7501 -c -g`

Here, you can specify the desired address and port with the `-a` and `-p` flags, respectively. The last flag should be either `-s` (for "set") or `-g` (for "get"), depending on the operation you wish to perform. Ensure that the peer you're trying to connect to is running before executing the test.

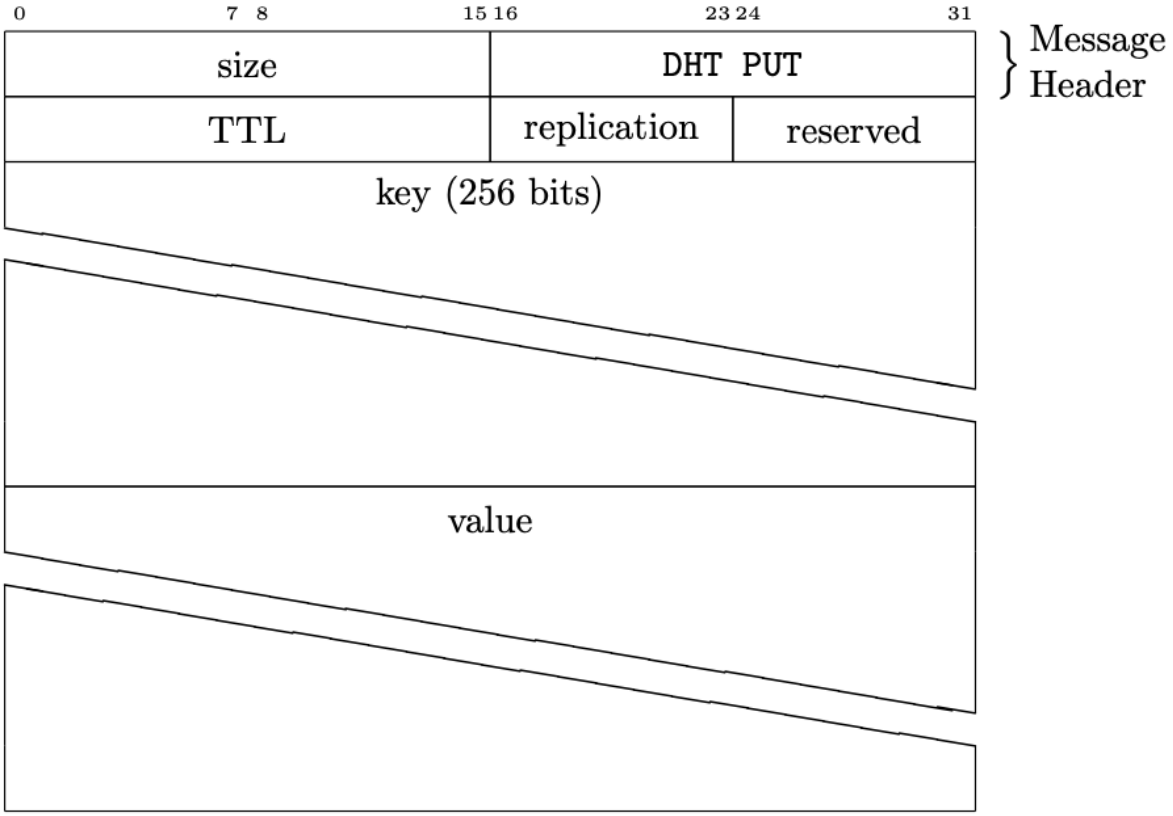
2.3. Documentation

You can also check documentation for further information and details. We used MkDocs to generate documentation by running the following in the project directory:
cd documentation && mkdocs serve

3. Peer-to-Peer Protocol

3.1. DHT PUT

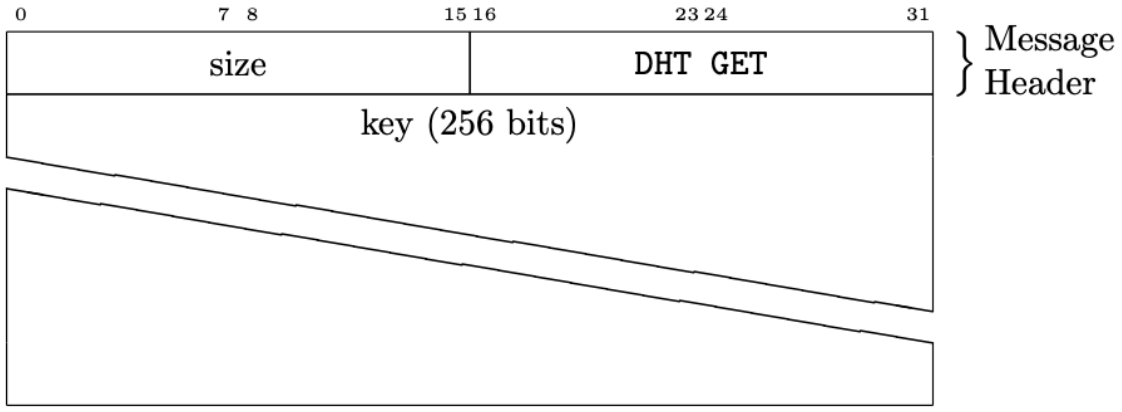
This message is used to store the provided key-value pair in the DHT.



- Size: Size of the body of the of the DHT PUT with the message headers
- Type: Type of the message (DHT PUT)
- TTL: Time to live in seconds given key value pair should be stored in the network. Peer responsible for the storing following key value pair will try to store it for the provided TTL if possible
- Replication: Number of copies should be done for the following key-value (over different peers)
- Reserved: Reserved space for extra messages

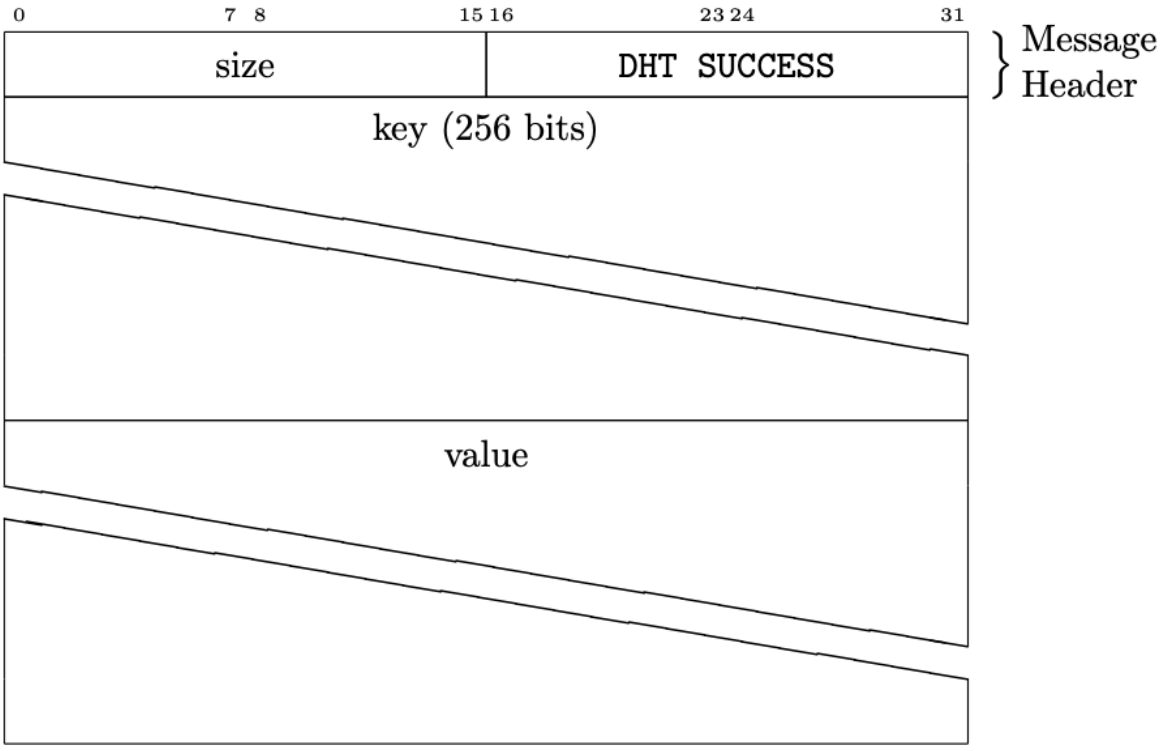
3.2. DHT GET

This message is used to ask the DHT method to search for a given key and provide the corresponding value, if it can be found in the network.



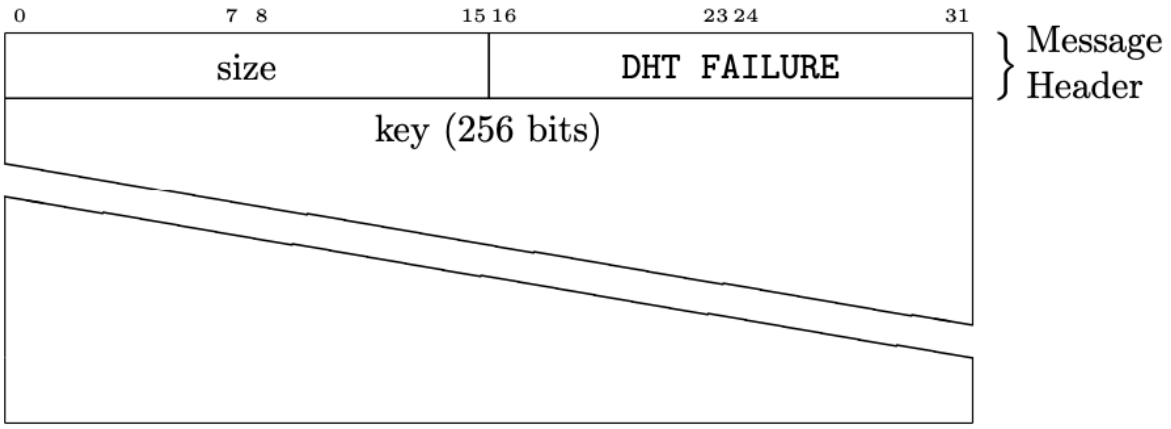
- Size: Size of the body of the of the DHT GET with the message headers
- Type: Type of the message (DHT GET)
- Body: Key of the key-value pair that should be retrieved from the network

3.3. DHT SUCCESS



- Size: Size of the body of the of the DHT SUCCESS with the message headers
- Type: Type of the message (DHT SUCCESS)
- Body: Body of the success message including key-value pair

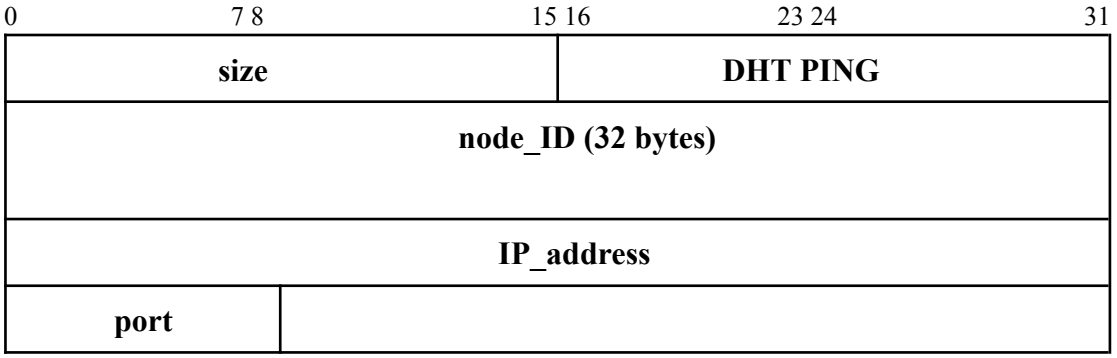
3.4. DHT FAILURE



- Size: Size of the body of the of the DHT FAILURE with the message headers
- Type: Type of the message (DHT FAILURE)
- Body: Body of the error message

3.5. DHT PING

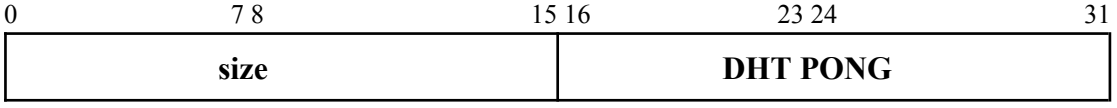
Ping is used to check if the address and the port is running and whether the peer is reachable.



- Size: Size of the body of the of the DHT PING with the message headers
- Type: Type of the message (DHT PING)
- node_ID: ID of the node that sends the message.
- IP_address : IP address of the node that sends the message.
- port: Port number of the node that sends the message

3.6. DHT PONG

Response to the “PING” message.



node_ID (32 bytes)	
IP_address	
port	

- Size: Size of the body of the of the DHT PONG with the message headers
- Type: Type of the message (DHT PONG)
- node_ID: ID of the node that sent the PING message (acts like nonce).
- IP_address : IP address of the node that sends the message (responder to PING).
- port: Port number of the node that sends the message (responder to PING).

3.7. DHT FIND_NODE

Find node message is used for the node lookups in the network. Sends requests to recently pinged nodes.

0	7 8	15 16	23 24	31
size		DHT FIND_NODE		
IP_address				
port				

- Size: Size of the body of the of the DHT FIND_NODE with the message headers
- Type: Type of the message (DHT FIND NODE)
- IP_address : IP address of the node that sends the message.
- port: Port number of the node that sends the message.

3.8. DHT NODE_REPLY

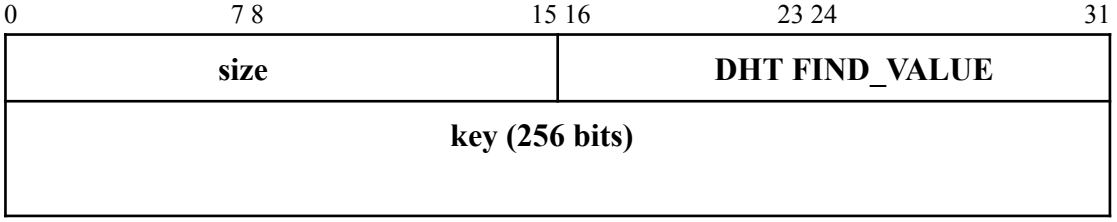
Response for the “FIND_NODE”.

0	7 8	15 16	23 24	31
size		DHT NODE_REPLY		
number_of_nodes		IP_address		
IP_address		port		
… rest of the nodes according to number_of_nodes				

- Size: Size of the body of the of the DHT NODE_REPLY with the message headers
- Type: Type of the message (DHT NODE REPLY)
- Number of Nodes: Count of nodes that Node Reply responded with. It contains information about nodes that matches Find Node message
- IP_address: IP address of the node
- port: Port number of the node

3.9. DHT FIND_VALUE

This message type is specifically for initiating the GET request searching process.



- Size: Size of the body of the of the DHT FIND_VALUE with the message headers
- Type: Type of the message (DHT FIND VALUE)
- key: Key of the key-value pair that should be searched within the network.

3.10. Service Functions

You can check the documentation for further details.

async def connect_node*(self, host, port, initiator)*: Creates the connection with the peers with the initiator parameter based on the role.
async def put_connection*(self, host, port, msg)*: Initiates the PUT request.
async def get_connection*(self, host, port, msg)*: Initiates the GET request.
def start_periodic_check*(self)*: Checking nodes for liveness. Utilizes 3 functions in the Service class.
async def put_service*(self, data)*: Starts to PUT service and sends key-value to the nodes.
async def get_service*(self, data)*: Starts to GET service and asks value from the nodes.
async def iterative_find_value*(self, key)*: Iterative method for looking for the value
async def query_node_for_value*(self, node, key)*: Checking whether the peer has the value or not.
async def handle_find_value_request*(self, key)*: Returns appropriate message to the node initiates the search.

4. Future Work

Below are possible areas for further development and improvement

4.1. IPv6 Support

As the internet starts using more IPv6 addresses, it's important for our system to keep up. We plan to add IPv6 support so that our DHT continues to work well in newer network setups.

4.2. Bootstrap Recovery Functionality

If the bootstrap node crashes, it's key to get it back up and running quickly. We're working on a way to help the bootstrap node recover faster after a problem, which will help keep data and operations stable.

4.3. Leader Election Mechanism

If the bootstrap node goes down, we need a backup plan. We're planning to add a leader election protocol. This will help keep things running smoothly and keep the network stable.

4.4. Enhanced Message Codes

Our current system works well for handling message codes, but we can make it even better. We plan to add special codes for different service functions. This will help us communicate and carry out tasks more efficiently.

By addressing these areas, we aim to make our Distributed Hash Table system more resilient, adaptable, and efficient, ensuring it remains at the forefront of DHT technology.

5. Workload Distribution

We worked well as a team and each brought our own skills to the project. Emin focused on coding, building services, and testing. His work made sure our system was strong and worked well. Can specialized in security issues and network protocols. He played a role as an architect, ensuring our project was both secure and intuitively designed.

We often met to share ideas and sometimes programmed together. This helped us stay on the same page and made the project better. We used one computer to manage our code on GitLab,

which helped us avoid technical problems. We think our mix of skills and teamwork really helped move our project forward.

6. Effort Spent

Most of our time was spent on coding and fixing problems to make sure everything was working right. We often coded together, which helped us both understand the work and made solving tough problems easier. For the final part of the project, workload was around 65% for coding, 15% for documentation and 20% for debugging.

We set clear goals for ourselves, similar to the Scrum approach. This helped us stay focused and meet the project's strict requirements.