P2P Systeme und Sicherheit (IN2194): Midterm Report

Mohamed Elzarei, Hady Mohamed

July 9, 2020

1 Architecture

We plan to design and implement our Distributed Hash Table (DHT) based on *Chord* protocol [1]. Our module is divided into two sub-modules each running on own thread: The API Module and P2P (DHT) Module and working in a producer-consumer manner.

1. API Module: Listens for incoming messages from clients (e.g DHT GET & DHT PUT messages) and relays them to the P2P module.

2. P2P Module: Consumes messages dispatched by the API module and implements the DHT protocol.

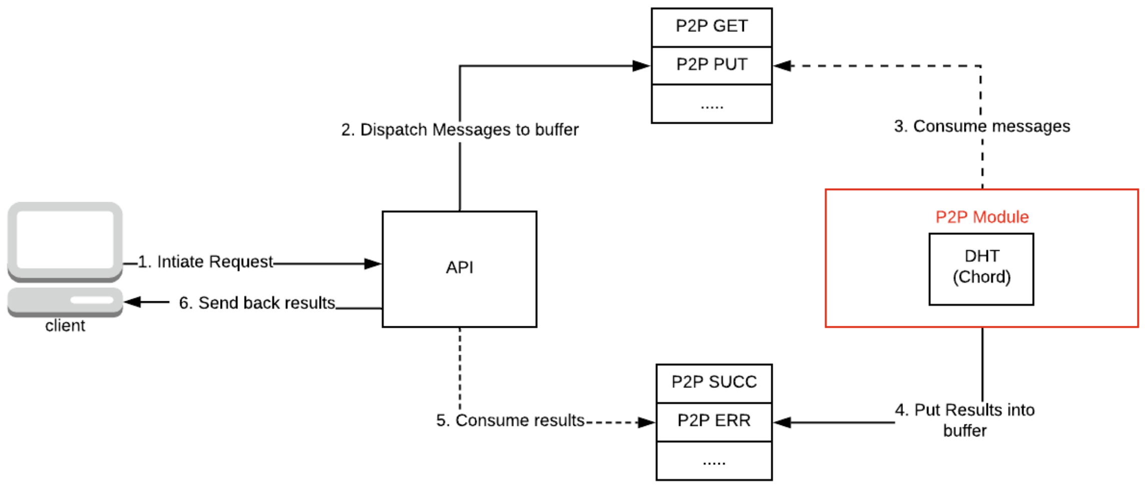

Figure 1: Request Flow

1.1 Implementation

1.1.1 Structure

Based on our initial report, we decided to use **Python**. We structure our code as sub-modules where each sub-module consists of several classes.

1.1.2 Process Structure & Networking

Each module runs in a separate thread and uses AsyncIO [2] for handling the requests.

1.1.3 Security

All our messages in the P2P protocol and API have the header format which we consider to be unique (If not so we can add a unique key to the header to correctly identify our protocol and application or do a handshake protocol but we leave this as future work for now).

2 P2P 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. This seemed more practical and adds an edge of efficiency at the expense of complete anonymity (ip address of the bootstrapped node has to maintained at each node).

2.0.1 Module communications

The current implementation expects all peer-to-peer communication of modules to happen over the same port.

2.1 PUT

The **DHT PUT** payload is encapsulated within this P2P PUT message.

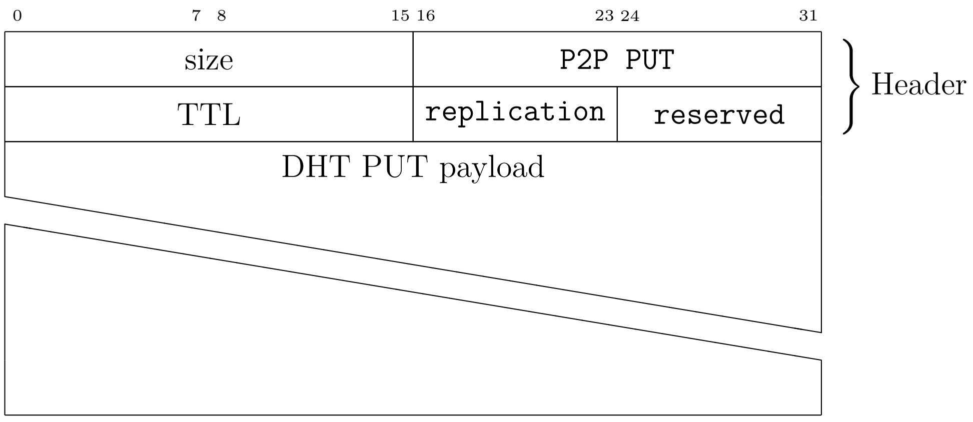

Figure 2: P2P PUT message

1. **Size**: The size of the DHT PUT (the payload / body) excluding any headers.
2. **Type**: The type of the message (**P2P PUT** in this case).
3. **Payload**: The payload of the message (The encapsulated **DHT PUT** payload).
4. **TTL**: Time in seconds the key-value pair should be stored in the network. (time to live) (best effort).
5. **Reserved**: reserved for use when needed.
6. **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).

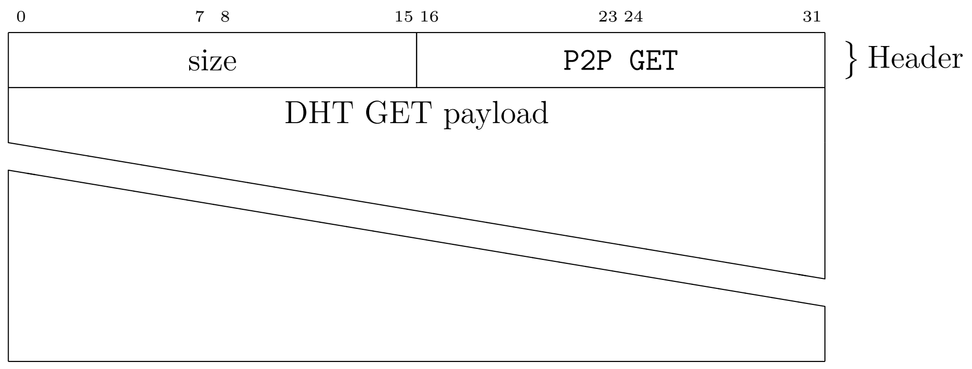

Figure 3: P2P GET message

2.2 GET

The **DHT GET** payload is encapsulated within this P2P GET message. (including key and headers)

1. **size**: The size of the payload (**DHT GET**)
2. **Type**: The type of the message (**P2P GET**).
3. **payload**: The body / content (**DHT GET** along with they key encapsulated + headers).

2.3 SUCCESS

A P2P message, which encapsulates a **DHT SUCCESS** message, after a successful retrieval.

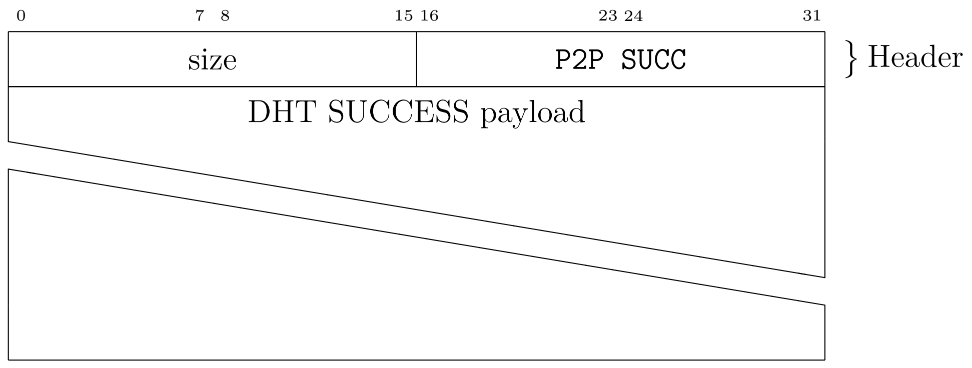

Figure 4: P2P Success message

1. **size**: The size of the payload (**DHT SUCCESS**)
2. **Type**: The type of the message (**P2P SUCCESS**).
3. **payload**: The body / content (**DHT SUCCESS** along with they key-value pair encapsulated).

2.4 FAILURE

A P2P message, which encapsulates a **DHT FAILURE** message, after a failed retrieval.

1. **size**: The size of the payload (**DHT FAILURE**)
2. **Type**: The type of the message (**P2P FAILURE**).
3. **payload**: The body / content (**DHT FAILURE** along with they key encapsulated).

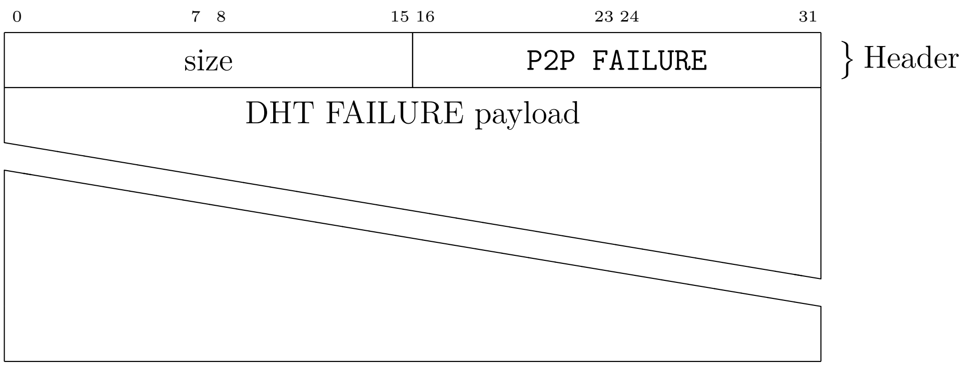

Figure 5: P2P Failure message

2.5 GET SUCCESSOR

Used to obtain the successor of a peer.

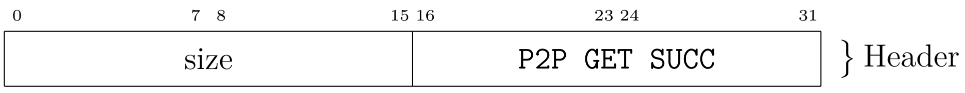

Figure 6: P2P GET SUCCESSOR message

1. **size**: The size of the message.
2. **Type**: The type of the message (**P2P GET SUCCESSOR**).

2.6 REPLY SUCCESSOR

A response to **GET SUCCESSOR** if it returns successfully.

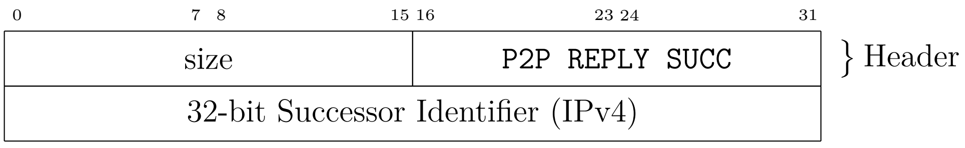

Figure 7: P2P REPLY SUCCESSOR message

1. **size**: The size of the message.
2. **Type**: The type of the message (**P2P GET SUCCESSOR**).
3. **payload**: IP the of the successor.

2.7 Exception Handling

In our implementation, The DHT does not make any guarantees and operates in a best-effort manner. Thus explicit error handling is not implemented. We rely on timeouts from clients and peers to retry the operation if no response is returned after a certain amount of time.

3 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. As it is very important for both of us to very well understand the protocol and project design. We plan to distribute the workload in the implementation of the protocol.

4 Future Work

- Implement custom field for protocol version and to uniquely identify our protocol.
- Explicitly handle errors in the protocols.

5 Effort Spent

The workload until this report was 20% for coding the API server as well as structuring the project. The other 80% was spent brainstorming and designing the protocol. We cant precisely bound the amount of time spent working on it though.

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.

[2] “Python asyncio.” https://docs.python.org/3/library/asyncio.html. Accessed: 2020-07-01.