VoidPhone Project **Final Report**
P2PSEC (IN2194) Team 45 - Rhodium
Prof. Dr.-Ing Georg Carle Benedikt Seidl, Stefan Su
Sree Harsha Totakura, Dr. Heiko Niedermayer Due Date: 08/10/18

1 Software Documentation

During the past summer term we implemented a distributed hash table (DHT) based on the Chord [1] as part of the VoidPhone Project. Our project is written in Rust [2] and has no dependencies on software outside of the Rust ecosystem. Therefore it is also platform independent and should compile on every major operating system. We have tested this for various Linux distributions and macOS. In the following, we elaborate on installation steps and on how to run our module, as well as the known issues of our release at the point of the deadline.

1.1 Dependencies

When you install the latest stable version of the Rust compiler[1], the *Cargo* build system and package manager is already included. Cargo allows to build the project and download all software dependencies automatically from the package repository *Crates.io* [3]. For this purpose, we have created a `Cargo.toml` file listing all packages (crates) and their required version required by our project. The list of the external crates and an explanation of the functionality of each package can also be found in Appendix B.

1.2 Installing and Running the Project

The project can be built by running `cargo build` in the project root directory. By default, a debug version is created but one can obtain further optimizations with the `--release` flag. One can also generate code documentation using `cargo doc`. For further information about the Cargo build process the Cargo Book [4] is a great reference.

Our project compiles to different targets. The `dht` binary starts a node in the peer-to-peer network and implements the api as defined in the project specification. Additionally, we provide an `api` target which is a small command line client that uses the specified api to get and store values from the distributed hash table.

To run the program, one can either execute one of the resulting binaries in the `target/` folder or use the Cargo command `cargo run`. The `--bin` argument must be set to either `dht` or `api`. Any command line arguments for the binary can then be appended after separating it with two dashes (`--`). In Table 1 all available parameters are described. Furthermore, in Appendix C some example commands are listed for starting a new network and joining a network using a bootstrap peer.

[1] `https://www.rust-lang.org/en-US/install.html`

Parameter	Purpose	Example
`-h, --help`	Prints help information	
`-q, --quiet`	Silence all output	
`-V, --version`	Prints version information	
`-v`	Level of verbosity	`v, vv, vvv, ...`
`-b <bootstrap>`	Address of a bootstrap peer	`192.168.0.1:31415`
`-c <config>`	Path to a custom config file	
`-t <timestamp>`	Print timestamps in log	`sec, ms, ns, none`

Figure 1: List of parameters

For testing convenience we created the a cli interface that enables us to send DHT PUT and DHT GET messages to our local peer. After running the `api` binary one can enter either "get" or "put" and provide the key and/or value to compose the desired message. The program will then send the message to the local peer and afterwards print the result on the command line. This script also requires a `-c` parameter for the path of the same config file as the local peer is using.

1.3 Configuration

The distributed hash table can be configured using a `config.ini` file which contains several parameters, some of which are required. As defined in the specification, the `listen_address` and `api_address` define the listen addresses for api and peer-to-peer connections. The other parameters are optional and provided with sensible defaults as described in table 2.

Option	Purpose	Default
`listen_address`	Listen address for peer-to-peer communication	-
`api_address`	Listen address for api connections	-
`worker_threads`	Number of threads handling peer-to-peer connections	4
`timeout`	Connection timeout in milliseconds	300000
`fingers`	Number of entries in the finger table	128
`stabilization_interval`	Time interval for stabilization in seconds	60

Figure 2: Config options

2 Inter-module protocol

We implement the Chord protocol as described in the paper by Stoica et al. [1] where the identifiers of keys and peers are located on an identifier circle based on consistent hashing. In the following section we define the different message formats used by our implementation.

Our message formats can be divided in two sections, according to the distinction we made in the introduction of this document. First we define the high level storage related message types which operate on direct connections to the targeted peer. After that we introduce our routing protocol messages which allow us to find nodes in the network efficiently by hopping over several other nodes on the way. The message type ids used for each of the defined messages can be found in Appendix A.

2.1 Storage messages

The storage protocol allows to store and receive messages from another peer. The contacted peer should only handle these messages if it is actually responsible for the provided key. Therefore, these messages can only be used if the IP address of the peer is already known.

The 256 bit raw key obtained from the api request is extended by one byte which contains the replication index. The identifier for the key is obtained by appending the replication index and hashing the resulting 33 byte array. This results in the same raw key with different indices being replicated to completely different positions in the P2P network. This makes it harder to gain control over every node storing this particular value.

2.1.1 STORAGE GET

This message can be sent directly to a peer which is responsible for the given key. The peer looks whether it has stored a value for the given key and returns it in a STORAGE GET SUCCESS message (see section 2.1.3).

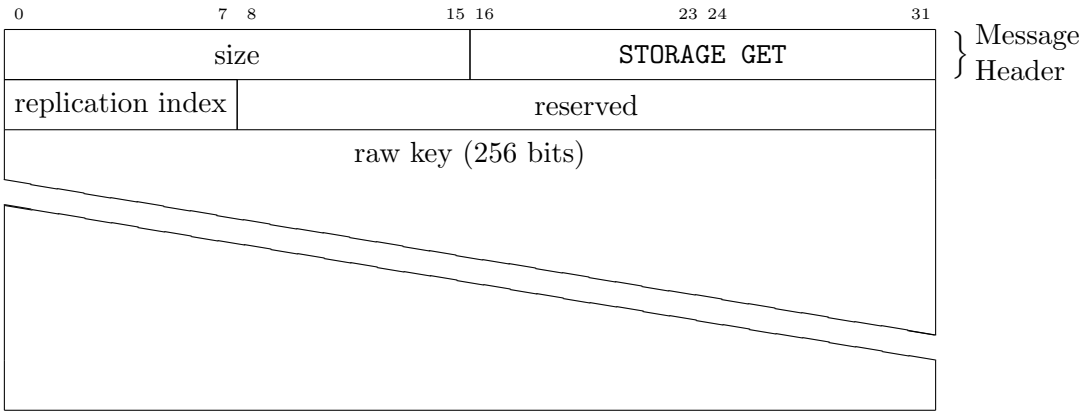

Figure 3: STORAGE GET message

2.1.2 STORAGE PUT

To store a message at a specific peer of which the IP address is already known, one can send this message. The peer should answer with a STORAGE PUT SUCCESS message (see section 2.1.4) if the operation succeeded.

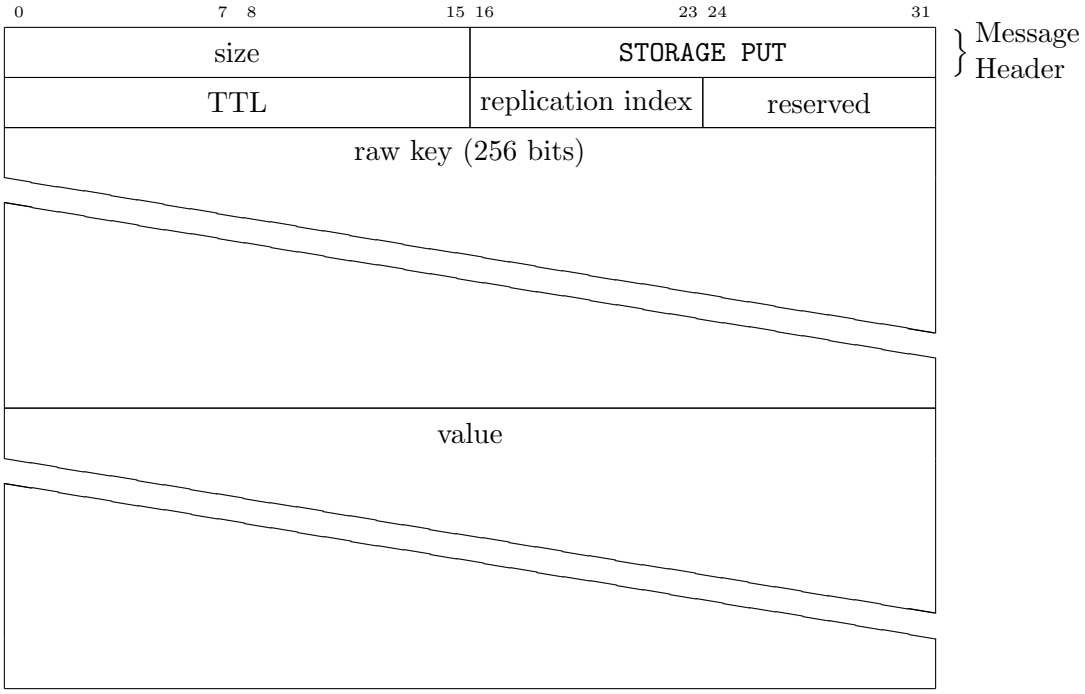

Figure 4: STORAGE PUT message

2.1.3 STORAGE GET SUCCESS

If after a STORAGE GET message (see section 2.1.1) the key was found, the peer should reply with the corresponding value attached to this message.

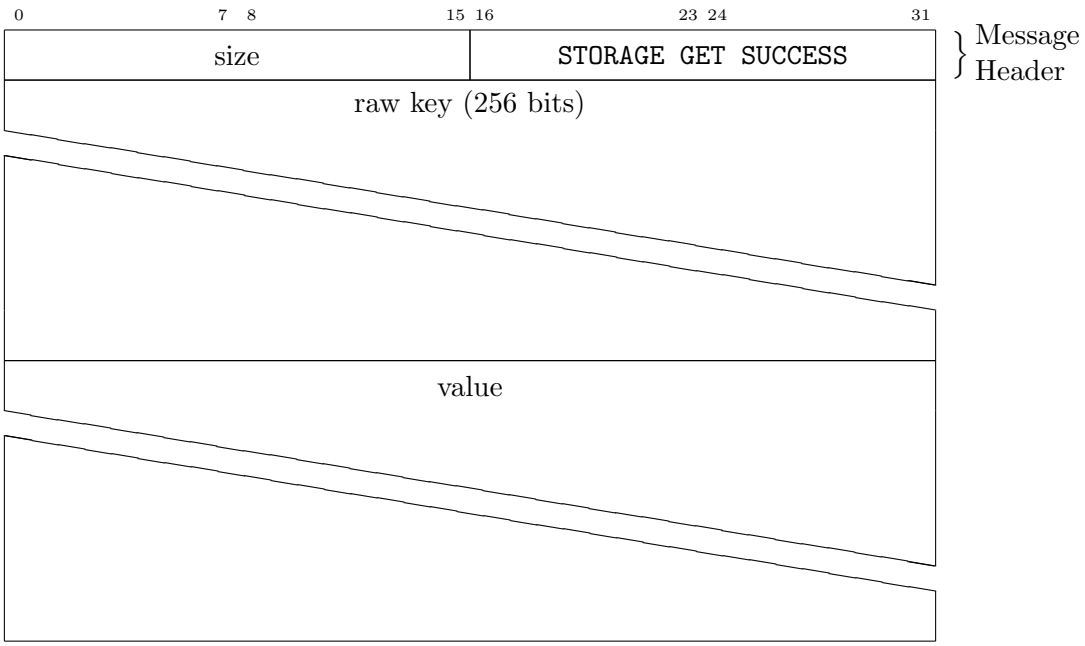

Figure 5: STORAGE GET SUCCESS message

2.1.4 STORAGE PUT SUCCESS

After a successful STORAGE PUT operation (see section 2.1.2), the peer should reply with this success message.

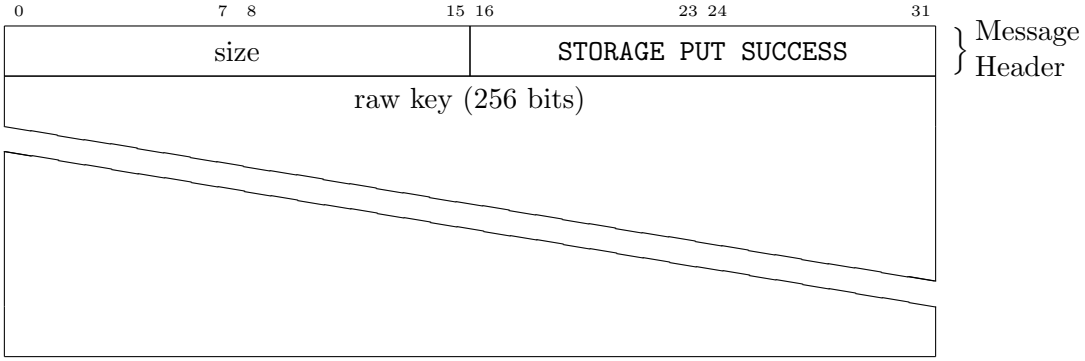

Figure 6: STORAGE PUT SUCCESS message

2.1.5 STORAGE FAILURE

If a STORAGE GET (see section 2.1.1) or STORAGE PUT (see section 2.1.2) operation fails for some reason, this message should be sent back. However, one cannot rely on a failure message being sent back since there can also be timeouts or other issues.

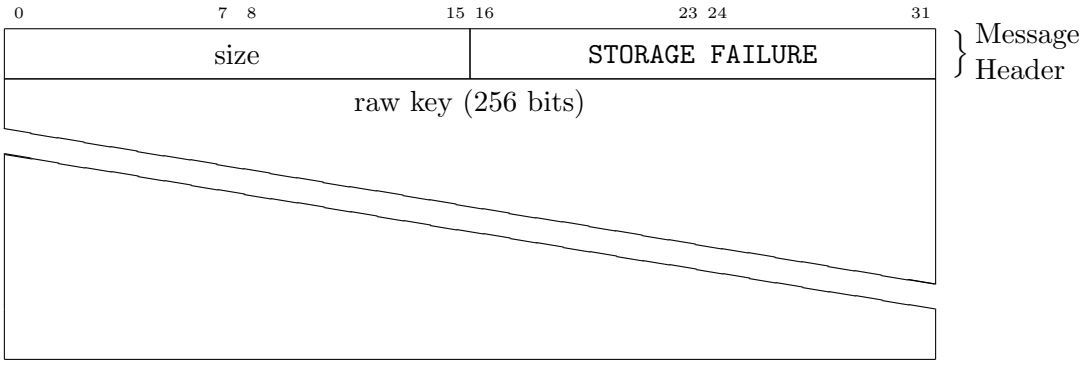

Figure 7: STORAGE FAILURE message

2.2 Routing messages

Since we use Chord as the underlying routing protocol, every peer has a unique location on the identifier circle obtained by hashing its IP address. It is responsible for all identifiers preceding it on the circle until the previous node.

To find a peer which is responsible for one given identifier, one has to ask other peers iteratively until the relevant peer has been found. This can be optimized by using a finger table which contains pointers into the identifier circle in exponentially growing distances. By following these fingers iteratively until another peer returns its own address, a result will be obtained in logarithmic time.

Additionally, the P2P network needs to be kept stable since new peers may join and other peers can drop out of the network. Therefore, every peer executes a stabilization routine regularly to update its finger table as well as its predecessor and successor pointer.

2.2.1 PEER FIND

This message initiates a lookup for a node responsible for the given identifier. The receiving peer is expected to reply with the known peer closest to the requested identifier on the circle using PEER FOUND (see section 2.2.2).

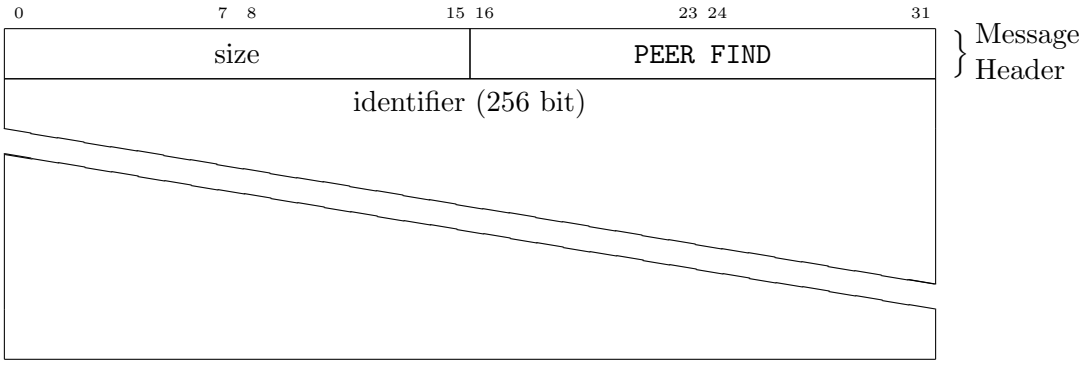

Figure 8: PEER FIND message

2.2.2 PEER FOUND

If, after a PEER FIND (see section 2.2.1) operation, a node has been found which is closest to the given identifier, the address of that peer should be included in this message. If the requested peer is responsible for the identifier, it will reply with its own address.

The address is always an IPv6 address. To represent an IPv4 address one can use the standardized IPv4-Mapped IPv6 address by appending the address to a string of 80 zeros and 16 ones, for example `::ffff:127.0.0.1` [5].

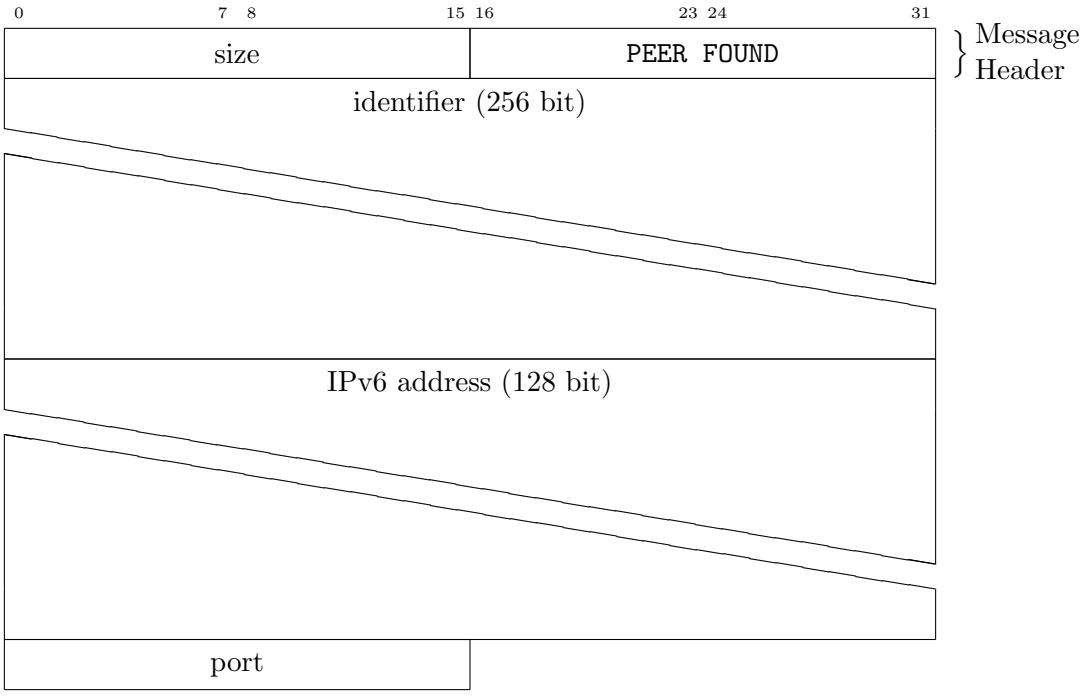

Figure 9: PEER FOUND message

2.2.3 PREDECESSOR NOTIFY

This message allows to notify some other peer of a potentially new predecessor. The receiving peer may use the given address to update its predecessor afterwards if applicable. IPv4 addresses are represented as described in section 2.2.2.

Furthermore, the peer is expected to answer with a PREDECESSOR REPLY message (see section 2.2.4) including the address of its current predecessor.

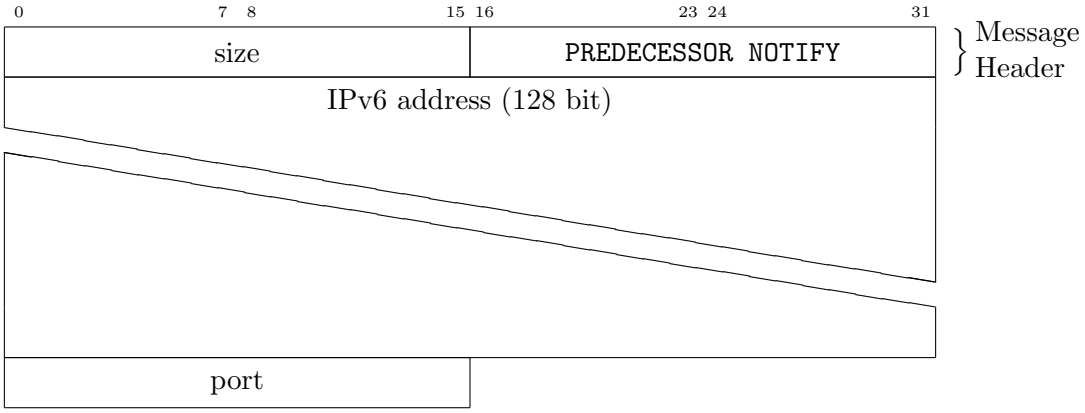

Figure 10: PREDECESSOR NOTIFY message

2.2.4 PREDECESSOR REPLY

When a peer receives a PREDECESSOR NOTIFY message (see section 2.2.3), it is expected to reply with this message including the address of its predecessor. IPv4 addresses are represented as described in section 2.2.2.

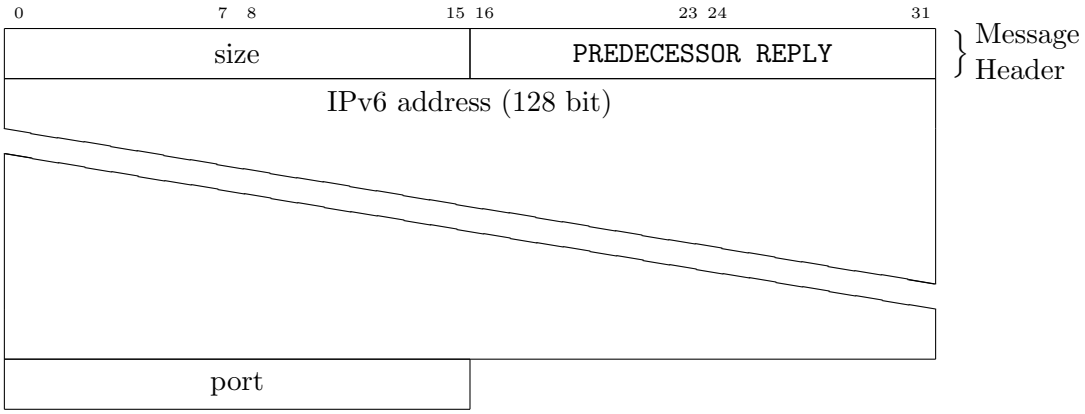

Figure 11: PREDECESSOR REPLY message

3 Limitations and Future Work

Since we had limited time to implement such a extensive project, there are some limitations in the current state of our implementation. In this section we explain which issues exist and which steps could be taken to eliminate them in a future version of our module.

3.1 Routing Stability

Our distributed hash table implements the defined behaviour as expected as long as the network remains stable and no churn occurs. However, there are currently no proper measures to handle node failures and peer leaving the network unexpectedly. This could be solved relatively easily by storing a successor list instead of just one successor. However, this requires an extension of the stabilization algorithm which we did not achieve in the given time period.

Furthermore, another issue is that on setting a new predecessor, a peer currently does not give that peer responsibility for the values stored in its identifier range. To achieve this, the peer should send STORAGE PUT messages to its new predecessor obtained from a PREDECESSOR NOTIFY request. This is also an easy to implement extension of our current code base.

3.2 Availability of Values

Currently our peers ignore the TTL field and store the values as long as they are part of the network. It should be easy to discard any values which exceeded their time to live. Furthermore, by storing the values not only on one peer but also on n of its successors, one can increase the probability that the value is available during the whole TTL and not lost before.

We do however respect the replication field and store as many copies of the value as required. Since we include the replication index during hashing to obtain the identifier, each replica is stored in a different part of the network. We could use this to improve lookup times for DHT GET requests by searching for different replication indices simultaneously instead of linearly.

3.3 Attack Resistance

To make the network more resistant against a possible Sybil attack, we should employ additional identity validation measures. It is already hard to get control over a whole identifier range since a peer's identifier is obtained by hashing its IP address. However, we could increase the cost of obtaining multiple identities by employing a challenge based on some proof of work during stabilization. Additionally, in order to counter possible Eclipse attacks, we can add proximity constraints to the finger tables to avoid filling it with nodes in similar IP ranges [6].

4 Work division and Efforts

We started our project by considering different protocols for distributed hash tables and in a joint decision opted to implement Chord. Together, we worked through the Chord paper [1] and designed our peer-to-peer messages accordingly. We then divided the implementation workload in smaller units and distributed them among ourselves.

Benedikt took care of designing the foundations of the application architecture and routing data structures and Stefan implemented the message parsing and serialization. Afterwards, we were able to work on the peer-to-peer interface (Benedikt) and api endpoint (Stefan) independently. However, each code change was first done in a feature branch so that we could review each other's code in granular merge requests.

References

[1] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM '01. New York, NY, USA: ACM, 2001, pp. 149–160. [Online]. Available: http://doi.acm.org/10.1145/383059.383071

[2] *The Rust Programming Language*, https://www.rust-lang.org/en-US/index.html, visited May 2018.

[3] *Rust Package Registry*, https://crates.io/, visited May 2018.

[4] *The Cargo Book*, https://doc.rust-lang.org/cargo/index.html, visited May 2018.

[5] R. M. Hinden and S. E. Deering, "IP version 6 addressing architecture," *RFC*, vol. 4291, pp. 1–25, 2006. [Online]. Available: https://doi.org/10.17487/RFC4291

[6] A. Singh, T. Ngan, P. Druschel, and D. S. Wallach, "Eclipse attacks on overlay networks: Threats and defenses," in *INFOCOM 2006. 25th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, 23-29 April 2006, Barcelona, Catalunya, Spain*, 2006. [Online]. Available: https://doi.org/10.1109/INFOCOM.2006.231

A Message Types

This enumeration defines the messages types. All numbers are in decimal system.

1000. `STORAGE GET`
1001. `STORAGE PUT`
1002. `STORAGE GET SUCCESS`
1003. `STORAGE PUT SUCCESS`
1004. `STORAGE FAILURE`

⋮

1049. (reserved until here for Storage messages)
1050. `PEER FIND`
1051. `PEER FOUND`
1052. `PREDECESSOR NOTIFY`
1053. `PREDECESSOR REPLY`

B External Crates

In the following we list all our dependencies from *Crates.io* as defined in our `Cargo.toml` file.

Crate	**Purpose**	**Version**
bigint	Provide 256 bit integer arithmetic	4.4.0
byteorder	Write integers in big or little endian	1.2.3
log	Provide logging with different verbosity levels	0.4.3
ring	Cryptographic operations based on *BoringSSL*[2]	0.12.1
rust-ini	Parse Windows `ini`-files	0.12.2
stderrlog	Log output generator using *stdout*	0.4.1
structopt	Ease the parsing of command line arguments	0.2.10
threadpool	Distributes work between different threads	1.7.1

Figure 12: List of external crates

[2] `https://boringssl.googlesource.com/boringssl/`

C Example Commands

After deciding for a bootstrap peer and distributing its public socket address to nodes who wish to join the network, the DHT module of the bootstrap peer can be started by running

To join an existing network using a bootstrap peer, the peer's socket address should be supplied using the `-b` parameter as follows.

```
$ cargo run -bin dht --
    -c <config-file>
    -b <socket-addr>
    -v <verbosity level>
    -t <timestamp>
```

Figure 13: Join a network using a bootstrap peer

If however no bootstrap peer is known and a new network should be started, one can just omit the parameter. Then no bootstrapping is performed and the routing information points to the peer itself.

```
$ cargo run -bin dht --
    -c <config-file>
    -v <verbosity level>
    -t <timestamp>
```

Figure 14: Start a new network without a bootstrap peer

The following example shows how to use the api command line interface tool. The config file provided by `-c` should be the same as the one used to start the distributed hash table on the current peer.

```
$ cargo run -bin api --
    -c <config-file>
```

Figure 15: Run the api cli