VoidPhone Project **Interim Report**

B2BSEC (IN2194) Team 45 - Rhodium
Prof. Dr.-Ing Georg Carle Benedikt Seidl, Stefan Su
Sree Harsha Totakura, Dr. Heiko Niedermayer Due Date: 06/08/18

1 Introduction

In this document, we describe the planned design and implementation of our distributed hash table (DHT) module for the VoidPhone Project. A DHT allows to store key-value pairs in a distributed network of peers. No peer has to store the whole hash table but every node in the network provides some storage and computing capacity to realize a functional system [1].

We distinguish between the api interface which is used to communicate with other modules on the same peer and the peer-to-peer or inter-module interface which allows the DHT modules on different peers to interact. The DHT provides two operations via its api interface, namely PUT and GET, which are used to store a value under a given key and later obtain the value for this key. The peer-to-peer protocol is more complicated and will be discussed in this document after introducing our architecture design.

2 Architecture Design

2.1 Application Architecture

To realize the distributed hash table we will implement the *Chord* protocol. The central aspect of Chord is to provide a distributed lookup method. This means to map a given key to a node in the network [1]. The important aspects are load balancing such that all nodes store approximately the same amount of data, failure handling and recovery as described in a later section and efficiency in the routing process.

On top of this core functionality, we can implement a file storage system which uses Chord to find one or several peers to store a file for a given key. By separating these to layers of functionality, we can keep our routing implementation as simple as possible and perform the file storage operations separately. This also allows to implement redundancy and error handling on a higher level.

2.2 Process Architecture

Our DHT implementation will be based on TCP for both the api interface and the peer-to-peer communication. Therefore, we listen on two ports given in the config for the two interfaces and wait for incoming connections in two event loops.

Each incoming request should be handled as fast as possible. Therefore we want to use parallelization to balance the load on several cores. This can be achieved by several means, for example starting several processes or using multi-threading. Using multiple processes makes sense when there is little communication needed between each process since inter-process communication is quite expensive.

However, since we need to work on shared memory between requests and also because we expect each request to only take very short to process, our preferred solution for parallelization is multi-threading. For this purpose, we use the thread pool pattern which creates a given number of worker threads and handles jobs from a job queue. Whenever a request reaches our

[1] We use “peer” and “node” as synonyms throughout this document. In the context of networks and graphs, we usually talk about nodes while on the implementation level they are called peers.

server, it creates a new task to handle this request and adds it to the queue. This allows us to work concurrently while not having the overhead of spawning too many threads.

3 Inter-module protocol

As already mentioned, we implement the Chord protocol as described in the paper by Stoica et al. [1]. In the following section we define the different message formats used by our implementation. After that, we explain which authentication and failure handling mechanisms we plan to implement.

3.1 Message formats

We divide our message formats 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 TCP connections. After that we introduce our lookup protocol messages which allow us to find nodes in the network efficiently.

The messages defined in this document are subject to change in case we want to make improvements to our design. In that case, we will explain our reasons in the final report.

3.1.1 STORAGE GET

This message can be sent to a peer which is responsible for the given key. Its ip address has to be known already. The peer looks whether it has stored a value for the given key and returns it in a STORAGE GET SUCCESS message 3.

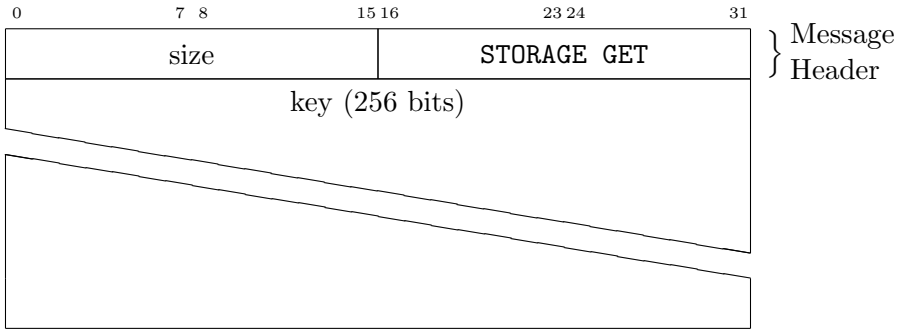

Figure 1: STORAGE GET message

3.1.2 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 4 if the operation succeeded.

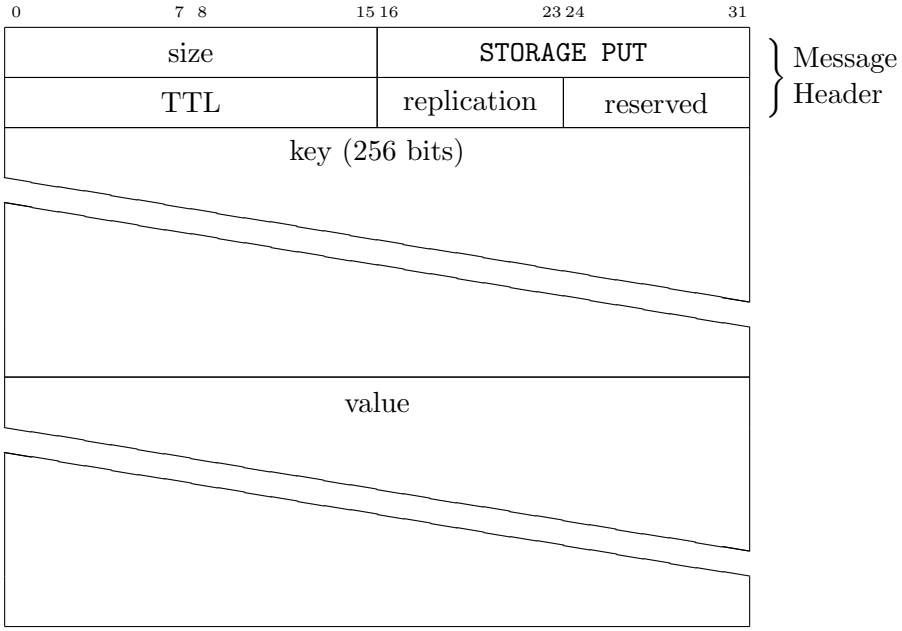

Figure 2: STORAGE PUT message

3.1.3 STORAGE GET SUCCESS

If after a STORAGE GET 1 message the key was found, the peer should reply with the corresponding value attached to this message.

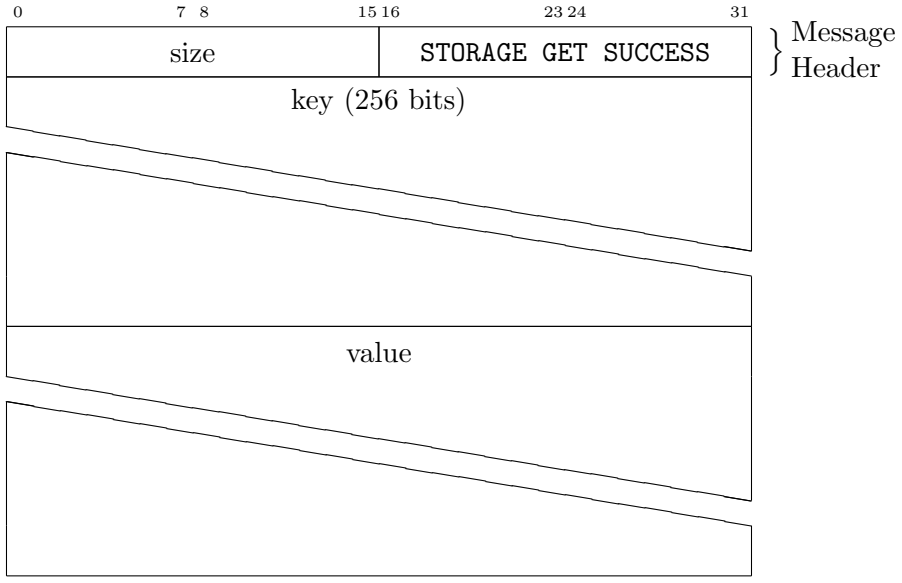

Figure 3: STORAGE GET SUCCESS message

3.1.4 STORAGE PUT SUCCESS

After a successful STORAGE PUT 2 operation, the peer should reply with this success message. The hash of the value should be appended to this message to ensure validity. It is still to be defined which hash function should be used.

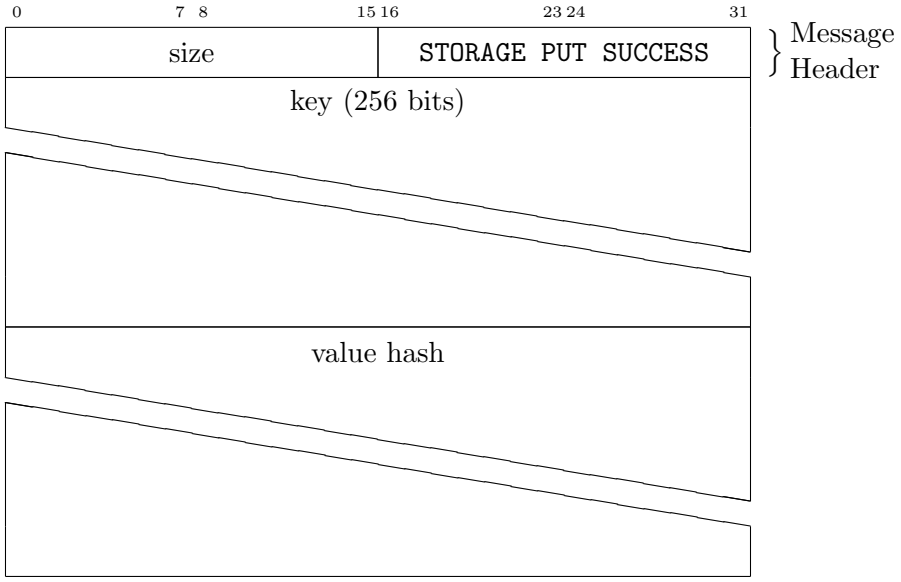

Figure 4: STORAGE PUT SUCCESS message

3.1.5 STORAGE FAILURE

If a STORAGE GET 1 or STORAGE PUT 2 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.

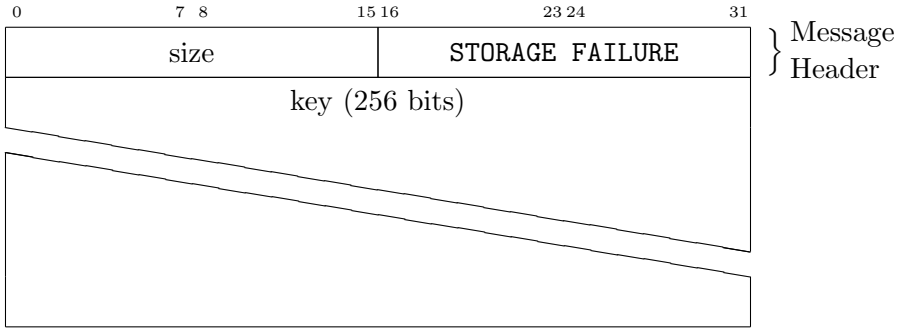

Figure 5: STORAGE FAILURE message

3.1.6 PEER FIND

This message initiates a lookup for a node responsible for the given identifier. If the receiving peer is not responsible for the ID, it is expected to forward this message to the next best node without modifying the *reply-to* field. This can be implemented using finger tables.

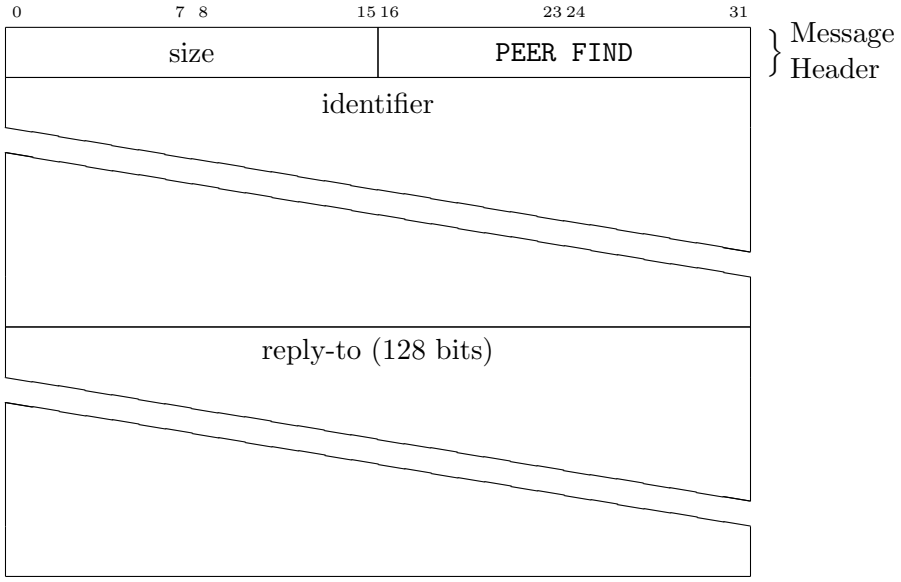

Figure 6: PEER FIND message

3.1.7 PEER FOUND

If, after a PEER FIND 6 operation, a node has been found which is responsible for the given identifier, that peer should reply with this message.

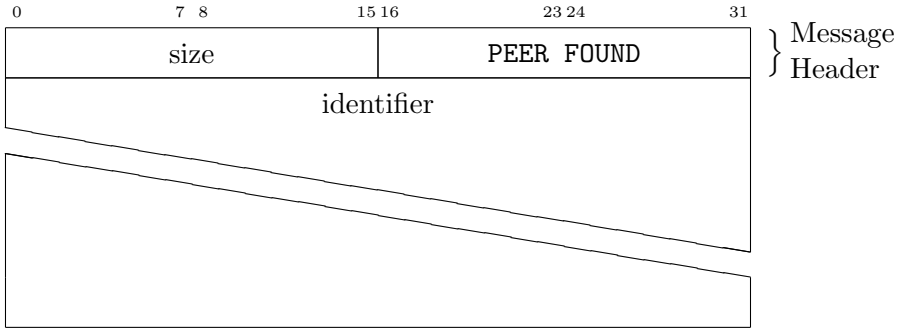

Figure 7: PEER FOUND message

3.1.8 PREDECESSOR GET

This message allows to query the predecessor of some other peer.

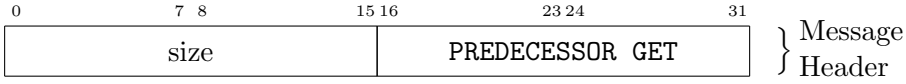

Figure 8: PREDECESSOR GET message

3.1.9 PREDECESSOR REPLY

When a peer receives a PREDECESSOR GET message, it is expected to reply with this message.

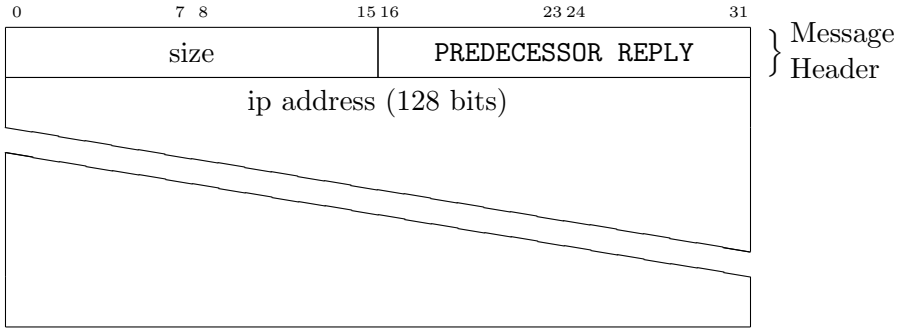

Figure 9: PREDECESSOR REPLY message

3.1.10 PREDECESSOR SET

To tell some peer about a new predecessor, this message can be used. The receiving peer is required to check whether it actually should update its predecessor value.

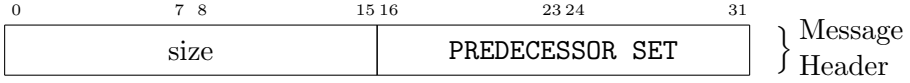

Figure 10: PREDECESSOR SET message

3.2 Authentication

When a new peer p joins the network, both the bootstrap peer and p need to verify whether they are running the correct software. There are several cases that may occur:

1. No module is running on the port that we appointed.
2. Another service is running on our DHT port that possesses uncompliant message formats.
3. Another service is running on our DHT port that just happens to use the same message types that we defined.

For our implementation we only need to actively mitigate case 3, as there are no consequences that would influence the behavior of our module in case 1 and 2 respectively. In case 1 we would not receive any response message and in case 2 we would not be able to process messages coming back from the foreign service anyway. Case 3 however would result in unpredictable behavior if the message types of the foreign service were to align ours and we would start processing unexpected payloads. This could be mitigated by establishing handshakes for every communication request. However, this is very expensive due to the messaging overhead. Alternatively, we can add a magic number field to our message format which should be unique to our application.

3.3 Failure handling

Chord's stabilization protocol protocol handles situations such as peers joining, failing or leaving the system. Maintaining correct successor pointers will guarantee the correctness of finding predecessors of a node. This is achieved by keeping a "successor-list" of the r nearest successors on the identifier ring. That way, "stabilize" will fix finger table entries and references pointing to the failed node. The successor list would also help with the objective of maintaining redundancy while storing key-value pairs by informing the storage layer of the state of the successors and therefore where the values need to be replicated to [1].

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

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 GET`
1053. `PREDECESSOR REPLY`
1054. `PREDECESSOR SET`

B Changelog

0.2: Update message type numbers (released 06/09/2018)
0.1: Initial report (released 06/08/2018)