# Midterm report

Martin Klapacz

July 4, 2023

## 1 Changes to assumptions in the Initial report

There are no specific changes to any of the plans or assumptions I have made in the initial report.

## 2 Module Architecture

### 2.1 Logical structure

The most important class in my project is the ChordService struct, which represents a node in the chord network. It implements all the RPCs I defined in `Chord.proto` file.

It contains the node URL and hash position identifier, that denotes its position in the CHORD hash ring. It also contains a Arc Mutex reference to a `FingerTable` struct instance which stores URLs and hash positions of other nodes in the cluster according to the CHORD protocol [1]. `Arc<Mutex<FingerTable>>` is a atomic reference-counted smart pointer to a mutex that contains a Fingertable instance. This guarantees thread- and memory-safety when multiple threads are reading and updating the finger table.

Another entry in the `ChordService` struct is the predecessor. This is essentially equivalent to an entry in the finger table. According to the CHORD protocol each nodes needs to keep a reference to its predecessor. We use the CHORD protocol to implement a distributed hash table. So every node also keeps a reference to a key value storage. The type of this field is `Arc<Mutex<dyn KVStore + Send>>`, which essentially is an reference to "something that implements" the KVStore trait. KVStore is an key-value-storage interface. Right now as the project is not finished yet, I only use a `Hashmap` wrapper named `HashMapStore`. That's a non-persistent storage, which is fine during development and debugging.

### 2.2 Process architecture

I designed the process architecture in such a way that there is a main function that starts up all other threads and awaits them. This happens in the main function of the `main.rs` file. This makes adding further threads and features pretty straight forward.

#### 2.2.1 Setup thread

When taking a look at the main.rs, you can see that the first thread that is started is the setup thread. This node is responsible for creating the finger table, key value storage and the predecessor entry for the node according to the CHORD protocol. It gets the info wether we this node starts up a new or joins an existing cluster. In the latter case this thread is responsible for getting the routing information it needs from the other nodes in the cluster. As a last step it sends the finger table and key value storage references to the gRPC thread.

#### 2.2.2 Tcp Client Thread

Another thread is the tcp client thread which handles client connections and implements the tcp API defined in the specification document of the P2P course. For every incoming client connection it creates a separate thread that handles this client and then, again, waits for another client connection. These client threads know the URL of the local gRPC service and delegate look-up requests and put/get operations send by the client.

### 2.2.3 gRPC Thread

This thread blocks until it receivs the finger table, key value storage and predecessor infos from the setup thread. Then it starts the gRPC server and is ready to answer gRPC requests from the local tcp thread or other remote nodes.

### 2.2.4 Shutdown Thread

This thread is called when sending a SIGINT signal to the main thread. It is responsible for saving the KV data to the predecessor before finally shutting down. It is, however, now fully implemented.

## 2.3 Networking

Generally speaking the project provides two distinct interfaces, The tcp interface for clients outside the cluster and the gRPC interface for node-to-node communication inside the cluster. The tcp endpoint receives client requests, validates them, and uses a gRPC client to forward the request to the local gRPC server. More precisely, it calls the CHORD look-up algorithm on the hash of the request key to get the responsible node for that key. Then it calls the actual GET or PUT RPC on the responsible node and returns the response with the correct format to the tcp client.

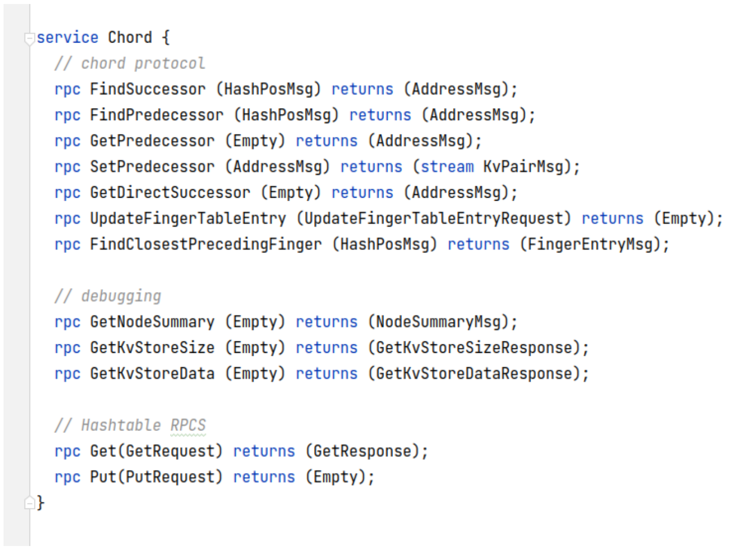
For thread-to-thread communication on a single node I use the Tokio asyncIO runtime library. An example of thread-to-thread communication is the transfer of the fingertable from the setup thread to the gRPC thread

## 2.4 Security measures

Up to now, I have not implemented any network related security measures. One thing one might consider a security measure is the fact that all critical member non-constant fields of the `ChordServer` struct are stored in mutexes available behind atomic reference-counted references (Arc). This prevents data races and data leaks at runtime.

# 3 P2P protocols - Message formats

The `Chord.proto` file serves as a contract, that defines the gRPC interface. The API is defined as follows.



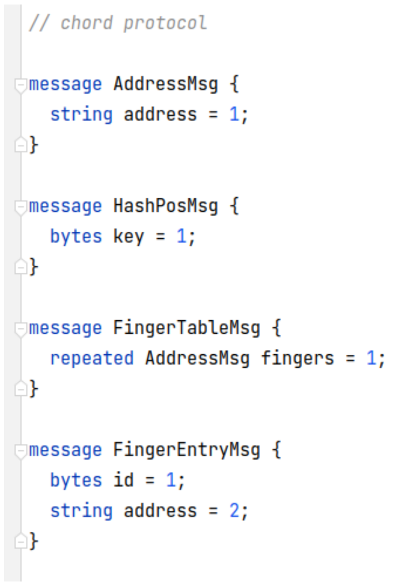
```
service Chord {
  // chord protocol
  rpc FindSuccessor (HashPosMsg) returns (AddressMsg);
  rpc FindPredecessor (HashPosMsg) returns (AddressMsg);
  rpc GetPredecessor (Empty) returns (AddressMsg);
  rpc SetPredecessor (AddressMsg) returns (stream KvPairMsg);
  rpc GetDirectSuccessor (Empty) returns (AddressMsg);
  rpc UpdateFingerTableEntry (UpdateFingerTableEntryRequest) returns (Empty);
  rpc FindClosestPrecedingFinger (HashPosMsg) returns (FingerEntryMsg);

  // debugging
  rpc GetNodeSummary (Empty) returns (NodeSummaryMsg);
  rpc GetKvStoreSize (Empty) returns (GetKvStoreSizeResponse);
  rpc GetKvStoreData (Empty) returns (GetKvStoreDataResponse);

  // Hashtable RPCS
  rpc Get(GetRequest) returns (GetResponse);
  rpc Put(PutRequest) returns (Empty);
}
```

The RPCs are separated in three sections:

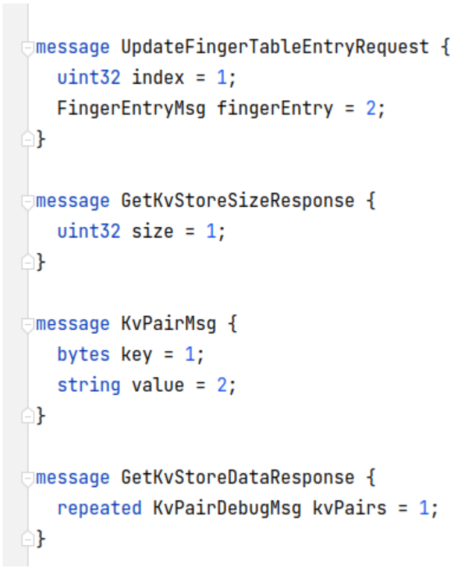
1. The RPCs that are required by the CHORD protocol
2. The RPCs that I use for debugging during development
3. The non-CHORD-related RPCs that implement the hash table API

## 3.1 CHORD messages

The CHORD related protobuf messages look as follows:



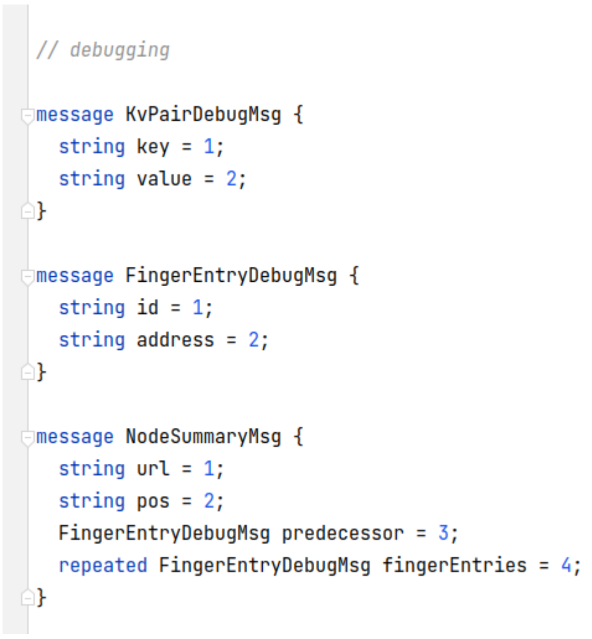
```
// chord protocol

message AddressMsg {
  string address = 1;
}

message HashPosMsg {
  bytes key = 1;
}

message FingerTableMsg {
  repeated AddressMsg fingers = 1;
}

message FingerEntryMsg {
  bytes id = 1;
  string address = 2;
}
```



```
message UpdateFingerTableEntryRequest {
  uint32 index = 1;
  FingerEntryMsg fingerEntry = 2;
}

message GetKvStoreSizeResponse {
  uint32 size = 1;
}

message KvPairMsg {
  bytes key = 1;
  string value = 2;
}

message GetKvStoreDataResponse {
  repeated KvPairDebugMsg kvPairs = 1;
}
```

Note that a `FingerTableMsg` only stores the URLs of the nodes referenced in this table. That's because we can always get the hash position of a node by hashing its address. So it is often sufficient to send only the addresses as the receiver can always compute the respective hash position of the nodes.

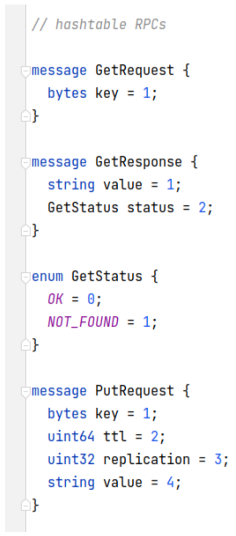
## 3.2 Debugging messages



```
// debugging

message KvPairDebugMsg {
  string key = 1;
  string value = 2;
}

message FingerEntryDebugMsg {
  string id = 1;
  string address = 2;
}

message NodeSummaryMsg {
  string url = 1;
  string pos = 2;
  FingerEntryDebugMsg predecessor = 3;
  repeated FingerEntryDebugMsg fingerEntries = 4;
}
```

The fields of the debugging messages are always of type `string` to make them human readable, when calling those procedures using Postman.

### 3.3 Hash table messages



```
// hashtable RPCs

message GetRequest {
  bytes key = 1;
}

message GetResponse {
  string value = 1;
  GetStatus status = 2;
}

enum GetStatus {
  OK = 0;
  NOT_FOUND = 1;
}

message PutRequest {
  bytes key = 1;
  uint64 ttl = 2;
  uint32 replication = 3;
  string value = 4;
}
```

These remote procedure calls are expected to be called on the correct node, i.e. the node that is responsible for the key, otherwise an error with appear. So a look-up call must a put or get call.

## 4 Future Work

The following aspects are still yet to be implemented:

1. The cluster join operation is in some cases not working correctly. I still have to fix it.
2. I want to add authentication mechanisms using certificates to the project.
3. I need to implement the CHORD stabilization mechanism
4. Add at least one persistent key-value storage
5. Replication

## 5 Workload distribution

The entire current implementation has been implemented alone by Martin Klapacz.

## References

[1] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: a scalable peer-to-peer lookup protocol for internet applications," *IEEE/ACM Transactions on Networking*, vol. 11, no. 1, pp. 17–32, 2003.