Documentation

Martin Klapacz

September 3, 2023

1 Installation and Startup

This installation is tested for Ubuntu 22.04. Running a node requires Rust (version 1.69.0) as well as the protobuf compiler to be installed.

1. Install Rust according to the documentation [1]
2. Install the Protobuf compiler:
   `sudo apt install -y protobuf-compiler`

1.1 Config arguments

Use the following command to start a node:

```
$ cargo run --package chord --bin chord -- -c <path-to-config-file>
```

The `configs` folder in the repository already contains 8 example config files. Replace `<path-to-config-file>` by `configs/config1.ini` to start up a new cluster. You can then use any of the configs to run nodes that join cluster.

Parameter	Description	Obligatory
`api_address`	Listen address for api clients	yes
`p2p_address`	Listen address for for other dht nodes	yes
`join_address`	Node address belonging to a cluster the node wants to join	no
`pow_difficulty`	Number of trailing bytes that need to be 0 in order to be a hash puzzle solution	no
`log_level`	Log level, possible values are off, error, warn, info, debug, trace	no

2 Module Architecture

2.1 Logical structure

The core of this project is the ChordService struct, which represents a node in the chord network. It implements all the RPCs which are defined in `Chord.proto` file.

It stores 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]. As defined in the Chord protocol the number of fingers in each table is equal to the number of bits in the key type.

`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.

[1] https://www.rust-lang.org/tools/install

Another entry in the `ChordService` struct is `predecessor`. According to the Chord protocol each nodes needs to keep a reference to its predecessor. Every node also keeps a reference to a key value storage. The current implementation uses a HashMap as an non-persistent in-memory storage.

2.2 gRPC and protobuf

Peer-to-peer communication is implemented by gRPC as a communication protocol. It is an open-source framework for robust service-to-service communication, built atop HTTP/2. Protocol Buffers (protobuf) serves as its default Interface Definition Language (IDL), facilitating strong typing and efficient data serialization. In Rust, the tonic library provides gRPC support, while prost handles Protocol Buffers. Both are built on Rust's asynchronous tokio runtime, offering a high-performance and type-safe way to build distributed systems.

2.3 Process architecture

As a gRPC server the `ChordService` does not act from its own. It provides only endpoints which can be called by clients. So in order to make the node run a for example a periodic stabilization process, predecessor health checking, etc. dedicated threads need to periodically call the respective endpoints on the gRPC service via a client. These threads do not interfere with each other and are only responsible for the feature they implement. I designed the process architecture in such a way that there is a main function that starts up the gRPC service thread and all other threads and awaits them. This happens in the main function of the `main.rs` file. In order to accomplish their task each thread gets a subset of the following parameters:

- A owned copy of the address to the local gRPC service, if it needs to communicate to the local gRPC service
- Smart pointers to directly read and write fields of the ChordService struct, reducing network load and avoiding adding a separate rpc call. Smart pointer copies are sent using Tokio's one-shot channels. Each thread gets an one-shot channel receiver for each reference it needs and thus gets a subset of the following references upon start up:
  - finger-table reference
  - key-value-storage reference
  - predecessor reference
  - successor-list reference

2.3.1 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. The thread either starts up a new cluster or joins an existing one. 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 smart pointers to finger table, key value storage, etc. other threads.

2.3.2 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 to the local gRPC thread.

2.3.3 gRPC Thread

This thread blocks until it receives 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.

2.3.4 Shutdown Thread

When performing a graceful shutdown (by sending a SIGINT) this thread comes into play. It transfers its entire local data in the kv storage to its successor node by calling the `handoff` rpc. This is a so called client-streaming-rpc which means that the node which is about to shut down sends its request, in this case the key-value-pairs, as a stream. This avoids loading the entire data into RAM before sending it to the receiver. This tremendously increases the efficiency of the handoff process.

2.3.5 Stabilization and FixFingers Thread

The Stabilization as well as the FixFingers thread are both an essential part Stabilization protocol implementation. They both periodically trigger the `stabilize` and the `fixFingers` rpcs. The `stabilize` rpc keeps the successor and predecessor handles in the node ring up to date. The `fixFingers` updates the finger entries in the node's finger table by calling `findSuccessor` for each fingertable in a round-robin fashion.

2.3.6 Health Thread

According to the Stabilization Protocol each node needs to periodically check if its predecessor is still reachable. If this happens to be the case, the node will set its predecessor handle to `None`. Which allows the stabilization thread to update the missing reference with the predecessor of the previous predecessor. The thread uses `health` rpc for that.

2.4 Successor List Thread

In order to make the whole cluster more robust, each node keeps a successor list. Right now the `SUCCESSOR_LIST_SIZE` is always set to 3. This thread keeps the local successor list up to date by periodically calling `get_successor_list` on the direct successor to update the successor list of the Chord instance with the successor list of the direct successor.

2.5 Networking

Each node project provides two distinct interfaces. The tcp interface (`api_address`) for clients outside the cluster and the gRPC interface (`p2p_address`) for node-to-node communication inside the dht 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 [2]. An example of thread-to-thread communication is the transfer of the finger table from the setup thread to the gRPC thread

3 P2P protocols - Message formats

The `Chord.proto` file serves as a contract, that defines the gRPC interface. The API is defined in figure 1.

The remote procedures can be separated into three different groups:

1. The RPCs that are required by the Chord and Stabilization protocol
2. The non-CHORD-related RPCs that implement the hash table API
3. Debugging related RPCs

```
service Chord {
  // chord protocol
  rpc FindSuccessor (HashPosMsg) returns (AddressMsg);
  rpc GetPredecessor (Empty) returns (GetPredecessorResponse);
  rpc GetSuccessorList (Empty) returns (SuccessorListMsg);
  rpc FindClosestPrecedingFinger (HashPosMsg) returns (FingerEntryMsg);
  // stabilization
  rpc FixFingers(Empty) returns (Empty);
  rpc Stabilize(Empty) returns (Empty);
  rpc Notify(NotifyRequest) returns (stream KvPairMsg);
  rpc Health(Empty) returns (Empty);
  rpc Handoff(stream KvPairMsg) returns (Empty);

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

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

Figure 1: Peer-to-peer gRPC api

```
// helper
message Empty {
}

// chord protocol
message AddressMsg {
  string address = 1;
}

message HashPosMsg {
  bytes key = 1;
}

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

message GetKvStoreSizeResponse {
  uint32 size = 1;
}

message KvPairMsg {
  bytes key = 1;
  string value = 2;
  uint64 expiration_date = 3;
}
```

Figure 2: Basic chord related messages

3.1 Chord protocol messages

The Chord related protobuf messages are shown in figure 3.1. `Empty` is used whenever a rpc does not require request or response arguments. The names of the proto messages are self explanatory.

Figure 3.1 lists messages which relate to the hash table feature of each node. A `GetResponse` always has a result type, which tells the client if the key exists, is not found or has expired.

```
// hashtable RPCs

message GetRequest {
  bytes key = 1;
}

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

enum GetStatus {
  GET_STATUS_NONE = 0;
  GET_STATUS_OK = 1;
  GET_STATUS_NOT_FOUND = 2;
  GET_STATUS_EXPIRED = 3;
}

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

Figure 3.1 shows messages used for stabilization as well for proof-of-work. Note that `NotifyRequest` contains a PowTokenMsg. A pow token contains a timestamp, the receiver can use to check if the token is still valid, a nonce and the pow difficulty.

Note that a `FingerEntryMsg` only stores the URLs and not the Ids 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.

[2] https://tokio.rs/

```
message GetPredecessorResponse {
    optional AddressMsg address_optional = 1;
}

message SuccessorListMsg {
  AddressMsg own_address = 1;
  repeated AddressMsg successors = 2;
}

message PowTokenMsg {
  uint64 timestamp = 1;
  uint64 nonce = 2;
  uint32 pow_difficulty = 3;
}

message NotifyRequest {
    AddressMsg address = 1;
    PowTokenMsg powToken = 2;
}
```

```
// debugging
message KvPairDebugMsg {
  string key = 1;
  string value = 2;
}

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

message NodeSummaryMsg {
  string url = 1;
  HashPosMsg pos = 2;
  FingerEntryDebugMsg predecessor = 3;
  repeated FingerEntryDebugMsg fingerEntries = 4;
  SuccessorListMsg successorList = 5;
}

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

The fields of the debugging messages in figure 3.1 are always of type `string` to make them human readable, when calling those procedures using tool such as Postman [3].

[3] https://www.postman.com/

```
// debugging
message KvPairDebugMsg {
  string key = 1;
  string value = 2;
}

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

message NodeSummaryMsg {
  string url = 1;
  HashPosMsg pos = 2;
  FingerEntryDebugMsg predecessor = 3;
  repeated FingerEntryDebugMsg fingerEntries = 4;
  SuccessorListMsg successorList = 5;
}
```

3.2 Testing

You can use the `validate_cluster.rs` binary for integration testing to validate the internal states of all nodes in a cluster. For example the command

```
$ cargo run --package chord
    --bin validate_cluster
    -- http://127.0.0.1:5601 http://127.0.0.1:5602 http://127.0.0.1:5603
```

validates a cluster consisting of three nodes running on the addresses listed arguments in the command. It checks if the finger table, successor list and predecessor handle of all nodes are correct.

3.3 Key-value-storage

Right now the system stores data in-memory using a HashMap of type `HashMap<Key, (Value, u64)>`. The `Key` type is a unsigned byte array of length 32. The value of the key used in the map is the hash of the key the client used in their PUT request. The values of the map are pairs of the `String` value and the time-to-live. The system does not actively check of TTL expiration. Instead whenever a client requests the value of a key, the node checks if the TTL value has expired. If yes, the node will delete that pair and notifies the client that the requested key has expired. Otherwise the client, receives its requested value.

3.4 Security Measures

3.4.1 Resource identity validtion through POW Tokens

POW (Proof of work) tokens are a possibility to combine an endpoint with proof of work in order to make sybill attacks more difficult. A POW Token has a timestamp and a nonce. The token is valid if and only if:

- The token is not expired
- The first n bytes of the hash of the timestamp concatenated with the nonce are zero

Right now, the `notify` endpoint requires a POW token. When calling this endpoint, the node will only process the call, if the token is valid. The token can be checked using the `.validate()` method. The stabilization protocol requires nodes to call `notify` periodically. This does not affect external tcp clients as `notify` is not called during Chord lookups. POW tokens therefore only increase the costs for stabilization and thus the costs for performing a sybill attack tremendously.
According to the Tokio documentation [4], it is not well suited for cpu-heavy parallel computation such

[4] https://tokio.rs/

as solving a hash puzzle. Therefore, I used the Rust library Rayon [5] to implement an fast multi-threaded solver for the hash puzzle. This makes sure the all of the node's computational resources are being used and a malicious node cannot easily increase the performance of the puzzle solver.

4 Future Work

The following features can be added in future versions.

1. Certificates and encryption. Data is sent between nodes without any encryption.
2. Right now replication is not implemented and just treated as a hint.
3. Depricated KV-pair cleanup in storage. In future versions a separate thread can periodically scan the local key-value-storage and delete all keys whose TTL field has expired.
4. The current implementation does not make use of the hostkey setting in the config file. This should be added in future versions.

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.

[5] https://github.com/rayon-rs/rayon