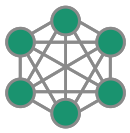
D7024E
Mobile and distributed computing systems



Project Camomile

Department of Computer Science,
Electrical and Space Engineering

Luleå University of Technology
Luleå, Sweden

Hjortsberg, Philip
phihjo-2@student.ltu.se

Wennerström, William
wenwil-5@student.ltu.se

Sladic, Edvin
edvsla-5@student.ltu.se

October 14, 2019

Contents

1 Technology choices

The requirements for a potential language is great concurrency support and one that is fast enough to handle thousands of concurrent network connections. Go has built-in support for CSP, Communicating Sequential processes, which utilizes channels to input and output data between processes. This simplifies the concurrent communication between the routing table, network and storage. The language was also recommended by the course coordinators.

The authors decided to mostly use the standard library. Potential data structures, such as linked lists, exists in the `container/list` package. The `net` package provides high-level UDP networking support through its `Dial`, `Listen` and `Accept` functions and the associated `Dial` and `Listener` interfaces. This was deemed enough for our use case.

Protobuf was picked for data serialization between the nodes. The protocol provides language neutral methods for serializing structured data. The serialized format is binary and more space efficient than other human readable serialization formats, such as JSON and XML. According to their own website, Protobuf is up to 100 times faster than XML[1]. Other serialization formats, such as FlatBuffers and Cap'n Proto, claimed[2][3] to be faster than Protobuf. However, the authors decided on Protobuf due to its higher popularity[4]. Protobuf also has a smaller wire format size in comparison with FlatBuffers[2].

The Kademlia paper specifies that SHA-1 is used as a cryptographic hash function to identify values. The authors decided to use Blake2B, which offers comparable, and even faster performance than SHA-1[5]. The size of Blake2B hashes can be changed to the 160 bit size specified in the Kademlia paper. However, the authors decided on using 256 bit keys instead. This is the smallest standard size of the hash in the Go library, `golang.org/x/crypto/blake2b`, and is the recommended smallest length to use[6].

The Zerolog logging framework is used for logging. It provides structured logging and better performance than the standard `log` library[7]. The logging can easily be disabled or redirected to a file with a non-blocking buffered writer that discards too slow writes. This ensures that the logging isn't impacting the performance of the implementation.

2 System architecture & design

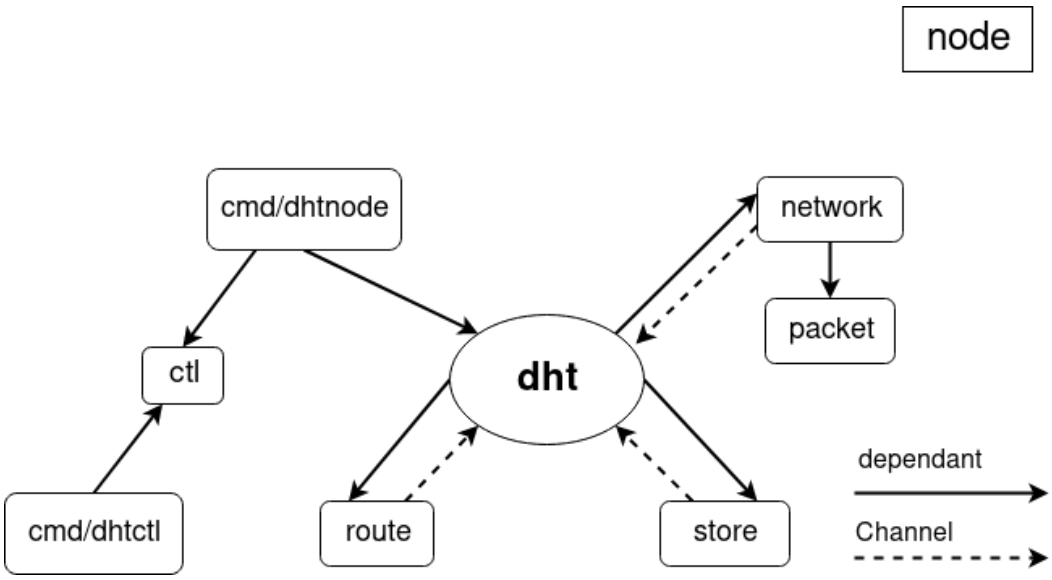


Figure 2.1: Overview of the system architecture of Camomile.

Camomile consists of two standalone programs, the node itself (`cmd/dhtnode`) and a CLI (`cmd/dhtctl`). The CLI communicates with a node, which holds the logic and implementation of Kademlia, using the RPC protocol.

The node is built up by the major packages as shown in the UML diagral in appendix A.1 and the system overview in figure 2.1.

Documentation for the project code and its packages and methods can be found under chapter 5 as a link to GoDoc.org.

2.1 Routing table

The bucket data structure is designed using an fixed size array as the main data structure which contains the k-buckets. These k-buckets uses a linked list to easily remove and add contacts. The closest contacts returned is contained in a short list, this short list uses an hash map to mimic a set data structure (existing duplicate items are overwritten as they share the same key), this ensures that duplicates cannot be inserted. Each contact contain the ID and an UDP address. The UDP address contains the IP address and the port that the contact listens on. The node ID and the distance are both fixed size slices, while a helper function calculates the bucket index by counting the length of the prefix (the number of leading ones in the distance).

The routing table instance, `route.Table`, provides a single channel that will be sent to when a bucket hasn't been touched in `tRefresh` (usually an hour) time, the `dht` listens on this channel and will issue a `FIND_NODE` for a random node ID that falls within the range of the outdated bucket.

The routing table implementation can be found in the `route` package.

2.2 Store

The internal storage of a node is designed around two hash maps, these are a key value type data structure efficient for storing large amounts of entries. One of the hash maps are dedicated to storing values from the network and the other is dedicated for values that this node has stored on the network and want to retain by republishing. Storing and accessing values is concurrent due to the use of mutexes in the add, retrieve and delete functions. Encapsulated are also two time based handlers, responsible for evicting expired values, republishing values that the node has stored on the network and replicating all the values stored on this node. The time based events that need to traverse the Kademlia network will communicate with the other parts of the program through a single channel. The Kademlia time constants and tickers are configurable to facilitate easier testing. In order to avoid hash collisions the group chose the Blake2B algorithm with a key size of 256 bits (32 bytes). The maximum value is limited to 1000 characters, this value is chosen with respect to the standard network packet MTU of 1500 bytes where 500 bytes are reserved for headers and Protobuf overhead.

The internal storage implementation can be found in the `store` package.

2.3 Network

The network follows the `Network` interface. This allows for several different network implementations to be used by the DHT, it also makes testing easier as the whole network can be mocked during development (see the `dht` package tests). The current implementation uses UDP, as requested by the lab specification. However, in the future, one could easily swap it with a TCP implementation.

To keep track of sessions, hash maps are used to store session IDs with their corresponding channel. This channel has a listener that waits for a response from another node. When a response is received from a node, the `network` package deserializes the raw packet and retrieves the session ID. This session ID is used to look up the awaiting channel in the hash map. The `network` package then sends the data structure to the channel retrieved from the hash map. The session only exists for one message and will be closed and removed after a single message has been sent through it. To handle time outs, the hash map has a Go routine that periodically checks for expired sessions according to a pre-defined TTL that is stored for each unique session in the hash map. If a session expires, a `nil` value is sent to signal the expiration for the receiver.

The network implementation can be found in the `network` package.

2.4 Packet

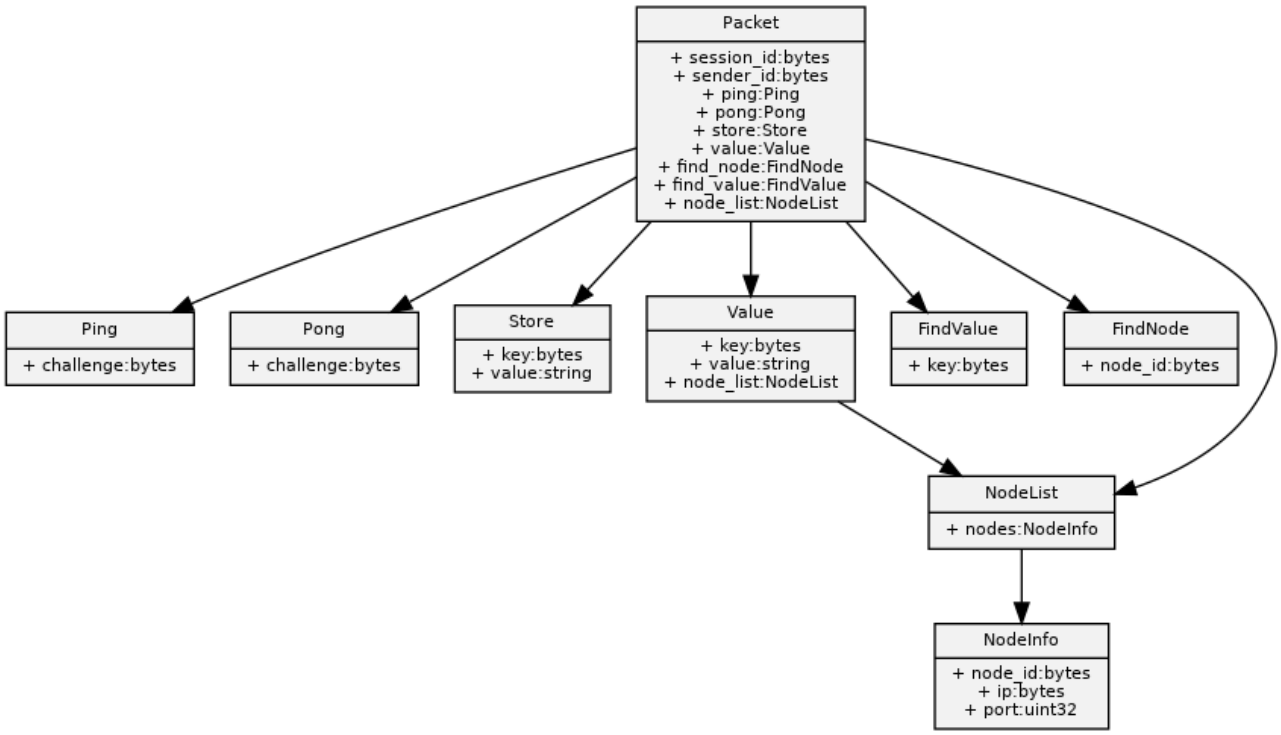


Figure 2.2: Protobuf packet design.

As mentioned earlier, Protobuf is used for message serialization. Each DHT call type is wrapped inside a main `Packet` message. This message wrapper always contains a session ID and a sender ID. The wrapped messages are then included using Protobuf's `oneof` type, which, as the name implies, enforces that one of the sub message types are included. See figure 2.2 for an overview.

The `Ping` and `Pong` message types are used for the `PING` call. The message types are the same except their names, they are separated as two different message types to help identify which is the initiator and responder.

The `FindValue` and `FindNode` message types are used for the `FIND_VALUE` and the `FIND_NODE` calls.

The Kademlia whitepaper specifies that the callee should respond with a list of closest nodes if it cannot retrieve the requested value for a `FIND_VALUE` call. This is the reason that the `Value` message type holds both a value and a node list. The `NodeList` can also be sent directly inside the `Packet`, this is what is sent as a response for a `FIND_NODE`.

The `Store` message type is used for the `STORE` call. The call is one shot and doesn't require any response message type. The reason for also sending the hash with a value is that it can be used to verify the integrity of the data when received.

2.5 DHT

The `dht` package is the central point of the node. The package imports the `network`, `store` and `route` packages (and `node`, which contain helpers for node IDs). All the calls supported by the DHT network is orchestrated from this package, such as `PING` and `FIND_NODE`. A `DHT` instance holds a `route.Table`, `store.Database` and a `network.Network`. All these packages are implemented to be used concurrently and can communicate back with the `DHT` instance using channels.

When the `DHT` instance is created, it will start one Go routine for each channel that the different packages above provides. For example, the `network` package provides a `FIND_NODE` channel and the `store` package provides a republish channel.

2.5.1 Walk

One of the most important parts of the `dht` package is the `walk` method. This is what traverses the DHT network to find either nodes or values. The method takes a single interface, called `Call`, this interface allows for several different DHT calls to use the same functionality that the `walk` method provides. The traversal is done according to what is outlined in the Kademlia whitepaper.

It begins with collecting the $\alpha = 3$ closest contacts in the local routing table, these are returned as a "shortlist" by the `route` package. The contacts are sorted according to their distance to the target node or value. Then, a iterative search for the target is initiated. Network calls will be issued to the α closest nodes. The call to be issued is defined by the `Call.Do` interface method that was provided as a parameter to the `walk` method. After the calls has been issued, the method will await the responses from all of the nodes, if any of the responses times out, the relevant contact will be removed from the shortlist.

Once all the results have been collected, the responding nodes are added to the routing table. Then, either the closest contacts that was in the response is added to the shortlist, or the traversal is stopped if any of the callees responded with a value. In that case, the value is returned to the caller of the walk using the `Call.Result` interface method.

The shortlist will be re-sorted, still with respect to the distance to the target. If the closest node is the same as the last iteration the walk will first do a call to all of the contacts still not contacted in the shortlist, and if no closer node was found, return the current sorted shortlist.

If a closer node was found, however, the walk will continue by contacting the next α nodes in the shortlist that has not been contacted yet.

2.6 CLI

The CLI is contained in a separate binary. The communication between the CLI interface and a node uses RPC over HTTP. The RPC callees with their relevant data structure definitions are kept in the `ctl` package. This package is imported by both the `cmd/dhtctl` and `cmd/dhtnode` programs. The CLI is issued by the running `cmd/dhtctl` program. The program provides five controlling flags,

- `-put` - Put a single value.
- `-get` - Get a single value by its Blake2B hash.
- `-ping` - Ping a node by its Kademlia/node ID.
- `-forget` - Forget a single value by its Blake2B hash.
- `-exit` - Terminate the controlled node.

There is also an optional `-address` flag that allows the CLI to control other nodes than the local.

Code that is related to the CLI can be found in the `cmd/dhtctl`, `cmd/dhtnode` and `ctl` packages.

2.7 REST API

The REST API is implemented using the `net/http` standard library. As the REST API is very simple, everything can be done using a single HTTP handler that handles each HTTP method using a switch-statement. The `dht` package can handle concurrent calls and can simply be passed as an instance to the HTTP handler. Instead of serving the endpoints at `/objects`, the authors decided to use `/` to save keystrokes. Furthermore, instead of answering with a `201 CREATED`, an `202 ACCEPTED` status code is returned instead as the value *may* still be processed by the DHT network. The authors deemed this to follow the REST principles more closely.

The REST API implementation can be found in the `cmd/dhtnode` package.

2.7.1 Reference

Method	Path	Form Fields	Header	Code	Description
GET	/{key}	N/A	Origin: {id}	200 OK	Retrieves a value by its hash key.
POST	/	value={value}	Location: /{key}	202 Accepted	Saves a value in the DHT network.
DELETE	/{key}	N/A	N/A	204 No Content	Orders the DHT network to forget a value.

3 Limitations & future improvements

Increase test coverage

At this moment the test coverage is at about 80%. However, the tests can be extended to cover even more of the source code. There is also room for improvements on making existing tests more extensive by writing more test cases for already test-covered code. he existing tests are however of good quality, and have been used continuously throughout the development process to ensure correct functionality on various parts of Camomile. The current code coverage can be found as a link in chapter 5.

Internal data structures

Camomile uses hash tables internally for the shortlists in the DHT. Performance could be improved by using fixed sized arrays instead as the internal structure for the shortlists. Since the shortlists are so small it really does not give an performance boost to use hash maps. This is largely due to that stop the world GC pauses are initiated on every insertion into the hash map, this can be seen when creating a call diagram of the `NClosest` benchmarks in `route/table_test.go`. It would make more sense to do a linear search than a hash map lookup and would at the same time be more effective on both memory space and execution time.

Camomile uses Protobuf serialization when communicating with other nodes on the network. As mentioned before, Protobuf is one of the more popular solutions for serialization. However, there are faster serialization options than Protobuf when it comes to performance, e.g. FlatBuffers. Using FlatBuffers would improve Camomile's marshaling and unmarshaling performance in terms of speed, but would lead to larger packets sent on the network. There are several options to elaborate with based on the desired characteristics of the serialization process. It might not be correct to say that neither Protobuf or FlatBuffers are better from a performance perspective, but that the serialization is possible to improve in the future based on the use-characteristics of the network.

Code duplication

There is some duplicate code in the network package. It would be a benefit to break out that code and make functions out of the duplicate snippets. Adding these functions would improve the maintenance of the code-base as well as the readability. Due to time constraints this has been of lower priority.

Benchmarks

Benchmarking could be used as a tool to identify hot spots in the source code and optimize parts of Camomile that may not be obvious intuitively. Extensive benchmarking tests has thus not been done, but would be a good future improvement for the development of Camomile. Note that both the `store` and `route` packages has already been partly benchmarked.

4 Log

4.1 Sprint 0

4.1.1 Issues

The following issues was the plan and backlog of action for *Sprint 0*. The section name describes the issue and the content describes what was accomplished with respect to this issue.

Add CI for unit tests and static analysis

A CircleCI environment that compiles, tests, statically analyzes and checks for race conditions is configured to run on every push to the repository. The environment also takes care of compiling the Protobuf specification. A link to the current build status can be found under chapter 5.

Decide on message serialization

Protobuf was decided for communication between the nodes. The CLI will also use Protobuf to control the nodes.

Implement protobuf

The initial message/packet structure has been implemented for all the message types. See figure 2.2 for a design specification.

Construct initial UDP network functionality

A simple UDP server and and client has been incorporated into the `dhtnode.go`. These are capable of starting and sending a simple message to the localhost.

Add documentation draft

Created a draft lab report with a link to the source repository at GitHub and *Sprint 0* log.

Write initial Dockerfile

The first revision of the Dockerfile is finished and works. This builds the `dhtnode` which is consisting of a Golang:1.13-alpine container with Protobuf and the dhtnode on it. A version of Docker Compose also exists which makes it possible to spin up several nodes at once.

Construct initial CLI

A simple set of flags was defined which should be available trough the CLI. These flags where implemented with the flag package of Go. At this early stage the CLI only takes the arguments specified and converts them to the correct type for the system to use.

4.1.2 Reflections

This sprint is to be considered successful due to the accomplishment of all requirements for *Sprint 0*. The group had gained a solid foundation of the Kademlia algorithm, an initial cluster of 50 nodes could be created, the plan for *Sprint 1* was written and based on the mandatory and qualifying attributes defined by the lab instructions and lastly this report was initiated. The initial thought of letting the CLI also communicate via Protobuf were also abandoned and replaced with the standard Go RPC library, due to unnecessary complexity for a simple task. No objectives neither mandatory or qualifying were fully fulfilled during this sprint.

4.2 Sprint 1

4.2.1 Issues

Send Protobuf message between two nodes

This was implemented by doing a `network` package that can be called by the `dht` package. The network can generate Protobuf packets and send them to a address by marshaling the packet before sending it.

The network package also listens for incoming packages on the specified Kademlia port. The package is also responsible for unmarshaling the packets and reporting the content of it back to the `dht` package via channels.

Implement Ping

The Kademlia Ping was implemented as two Protobuf message types, Ping and Pong. These messages are handled just like any other packet in Kademlia by the network. The `dht` package can initiate a ping towards a node but also listen for incoming pong requests. If a ping-pong is exchanged the buckets are updated accordingly in the respective nodes taking part. The challenge is generated by the sending nodes network package. This challenge is also evaluated when the pong is received.

Design & implement bucket data structure (routing table)

The routing table is implemented according to the design specified above. Most of the code has been tested using unit tests. Some benchmarking has also been done where it was discovered that the linked list data structure is a bottleneck that could be improved. Furthermore, the short list hashmap performs somewhat slow due to the small size and high number of insertions (each insertion will produce a stop the world GC pause). A simple fixed size slice would probably perform better.

Design & implement object data structure (storage)

The data store is implemented according to the thought out design. It has a complete suite of tests to ensure that it functions as intended. Some benchmarks has been created inorder to briefly evaluate the performance, the result was deemed satisfactory and no design rework was needed with respect to performance. The implemented functions so far support the Kademlia store and get. It also supports the following timer based Kademlia events; republishing values, eviction (deletion) of expired and non-republished items and replication of the complete database of items stored on this node. This component was later during the sprint connected to our `dht`.

Implement Kademlia core with network, route table and storage

The Kademlia core resides in the `dht` package and is the central point that implements the iterative functions, such as `iterativeFindNode` and `iterativeStore`. Each operations, e.g. `Join`, `Put` and `Get` uses these iterative functions to walk the DHT and find nodes, or find and store values. Channels are heavily used to communicate with the `network` package. The core uses parallelism to send up to α network calls concurrently. This concluded requirement **M1**.

CLI RPC implementation

Commending the nodes required a functioning CLI and was therefore prioritized amongst the developers. The CLI has support for most Kademlia commands, except forget. During the sprint, the group connected various CLI RPC calls to the `dht` package.

Streamline cluster creation

Running the docker containers turned out to be complicated with how we wanted to bootstrap the nodes as we wanted to use new random IDs on every run. To rectify this we created a bash cluster deployment script. It's possible to accomplish the same result with Docker Compose but the group deferred to known technologies in order to save time.

4.2.2 Reflections

The store turned out to have some implementation faults which led to not all timer based events working as intended. These errors will need to be corrected in *Sprint 2*. Two bugs were

found during the period of the sprint and corrected. At the end of the sprint the project fulfills the following requirements, **M1, M2, M3, M4, M5** and **U6**. This concludes that *Sprint 1* is compliant with the requirements for a higher grade, stated in the lab instructions.

4.3 Sprint 2

4.3.1 Issues

Connect storage republishing channel with dht

The implementation of republish got polished by reducing the amount of data needed to be sent in the republish channel. The channel where republish events is posted is now properly connected with the `dht` package, which in the end leads to a store request being sent on the network.

Storage replication

The hourly replication of the values stored had a small implementation error which made the replication replicate the value from the local data base instead of the one with values stored from the network. This was corrected and internally connected to the `dht` package.

Forget CLI command

The ability to let a node forget, or delete a value was implemented during this sprint. The forget command works according to the Kademlia specification by removing the value to forget from the nodes local data base and therefore it's no longer republished on the network and eventually not stored on any node.

RESTful application interface

In accordance with the qualifying objectives the group implemented a RESTful API from which a node can be controlled. This API supports Kademlia Get, Put and Forget by a somewhat logical mapping of HTTP to the methods `GET`, `POST` and `DELETE`.

Improve test coverage

In the beginning of *Sprint 2* the project only had a coverage of about 75%. The test coverage for the project has a goal of at least 80% in order to conform with the qualifying objective **U5**. At the time of writing, the coverage is 84%.

Implement network timeouts

There must be logic that handles network timeouts, at the beginning of the sprint all network calls were expected to respond and would therefore cause the node to hang in a waiting state. The network timeout implementation solved this as intended, if a node doesn't answer for a set amount of time the session is discarded.

4.3.2 Reflections

Apart from all the planned work items above, we also fixed some unexpected bugs. One of those being that any call would not work if a bucket containing the contacted address was full. We also corrected that buckets is refreshed correctly, and corrected a storage related bug. In the end of this sprint, we have achieved the last mandatory objective **M6** and the remaining qualifying objectives **U1, U2, U3, U4, U5**.

4.4 Final Statement on Objectives

The lab authors considers themselves to have fulfilled all the objectives, both mandatory and qualifying.

5 Links

- Git repository: https://github.com/optmzr/camomile
- Code coverage: https://codecov.io/gh/optmzr/camomile
- Documentation https://godoc.org/github.com/optmzr/camomile
- Build status: https://circleci.com/gh/optmzr/camomile

Bibliography

[1] Google Developers (2019) *Protocol Buffers - Why not XML?* Fetched 2019-10-14, from: https://developers.google.com/protocol-buffers/docs/overview#whynotxml.

[2] FlatBuffers (2019) *FlatBuffers: C++ Benchmarks* Fetched 2019-10-14, from: https://google.github.io/flatbuffers/flatbuffers_benchmarks.html.

[3] Cap'n Proto (2019) *Cap'n Proto: Introduction* Fetched 2019-10-14, from: https://capnproto.org/.

[4] Google Trends (2019) *Protobuf, capnproto, flatbuffers - Explore - Google Trends* Fetched 2019-10-14, from: https://trends.google.com/trends/explore?date=2018-10-13%202019-10-14&q=protobuf,capnproto,flatbuffers.

[5] BLAKE2 (2019) *BLAKE2 - fast secure hashing* Fetched 2019-10-14, from: https://blake2.net/.

[6] GoDoc (2019) *Package blake2b* Fetched 2019-10-14, from: https://godoc.org/golang.org/x/crypto/blake2b.

[7] GitHub (2019) *rs/zerolog: Zero Allocation JSON Logger* Fetched 2019-10-14, from: https://github.com/rs/zerolog.

A Illustrations

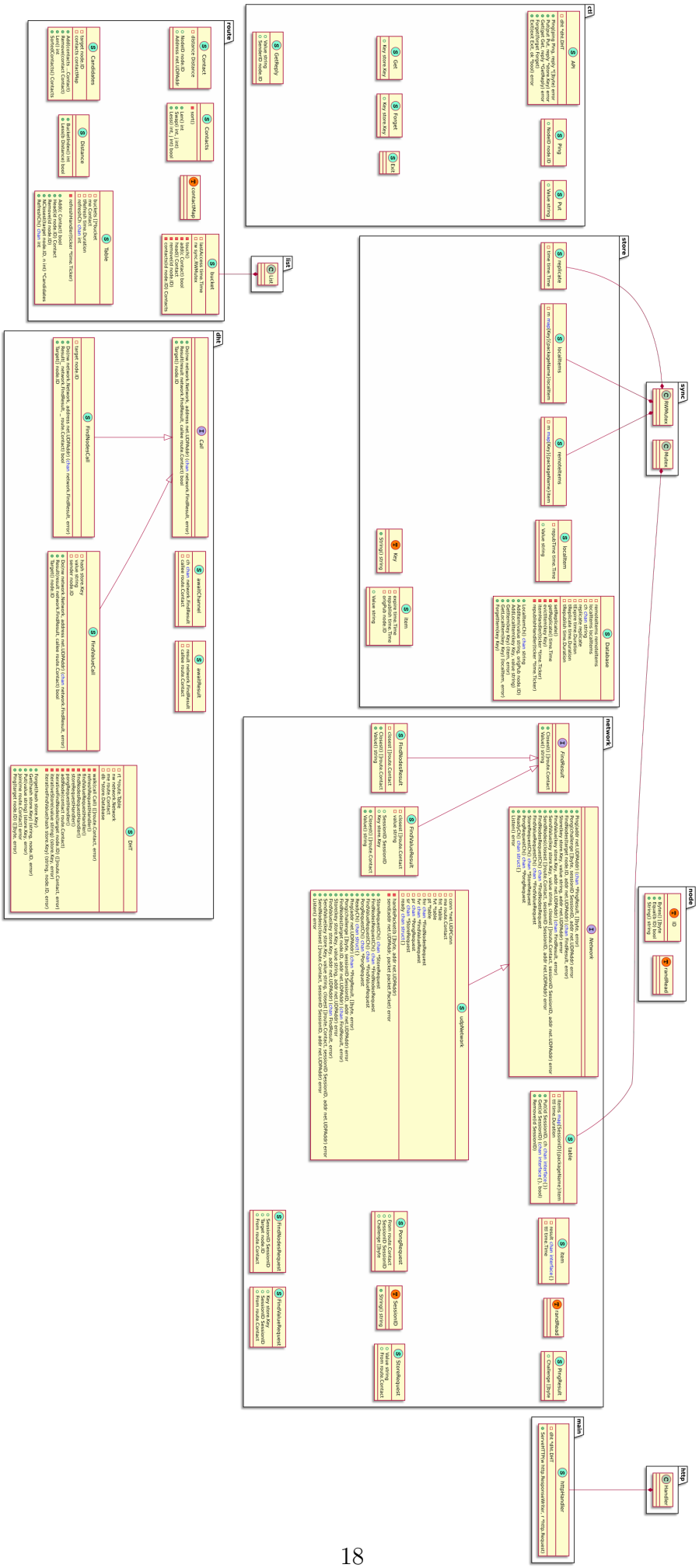


Figure A.1: Camomile UML diagram.