Midterm Report. DHT Peer-to-Peer Project

Denis Grebennicov

July 8, 2020

1 Changes to your assumptions in the initial report

- no real changes since everything went according to plan, at least for now

2 Architecture of your module

2.1 Logical structure (classes, . . .)

- TCPServer
  - tcp server which processes the messages according to specification
  - loads config and sets up the storageActor (basically DHT-node instance)
  - tcp server acts as a proxy/entry point into the system
    - decodes the incoming messages into the internal format
    - forwards the message to the storageActor
    - encodes the resulting message into tcp message response
- StorageActor
  - processes the DHT messages, like `DHTPut` and `DHTGet`
  - sends messages to corresponding aggregator actors (getter/setter actors)
- SetterActor
  - sends put messages to successor nodes
  - checks if the minimal number of successful writers had MutationAck for `DHPut` and sends this information to the storageActor

- GetterActor
  - sends get messages to successor nodes
  - returns the value which occurs the most
- RoutingTable
  - the child actor of the storageActor
  - maintains the routing information of the DHT node according to the Chord protocol
- FingerTable
  - class responding for logic related to fingerTable (e.g. finding the closest successor to the given id)
- FixFingersActor
  - helper actor which searches for the right node associated with the fingerTable entry
  - updates the table accordingly by sending this data to routingActor
- HeartbeatActor
  - helper actor sending the hearbeat messages to the predecessor and successor and sends this info to routingActor
- StabilizationActor
  - helpel actor responsible for stabilization process according to Chord algorithm
- WebService
  - web frontend
  - implemented as a centralized component in order to visualize and monitor the peer-to-peer system
  - maintains the information of the current alive nodes
  - if doesn't receive information from the alive node within 10 seconds, assumes node is dead

2.2 Process architecture (threading, multi-process, ...)

- every DHT node is a single JVM process
  - initially one process could span multiple DHT nodes
    - due to location transparency of the actors the migration was smooth
      - since local or remote, actor communicate via sending messages
    - could be still used for pseudo-nodes
- the actor communication is asynchronous and managed by the actorSystem thread pool

2.3 Networking (AsyncIO, goroutines, ...)

- as noted above the actor system is completely asynchronous
- managed by the internal actorSystem thread pool
- internal Akka scheduler to schedule between actors
- message processing is synchronous within one actor
  - i.e. actor processes one message at a time
  - yet, sending messages is a non-blocking activity

2.4 Security measures (encryption/auth, ...)

- currently no encryption/auth is implemented

3 The peer-to-peer protocol(s) that is present in the implementation

3.1 Message formats (as given in this document)

- all the specified messages are supported and tested
- other internal messages are used, yet they (currently) use the JVM encoding due to ease of use (just define a case class and you can send it with manually implementing encoder/decoder)

3.2 Reasoning why the messages are needed

- various reasons, e.g.
  - retrieving value for key from the node without sending `DHTGet` message to the successor nodes
    - basically a message for accessing the hash table and returning the result
  - similar logic for `DHTPUT`
  - all sorts of routing messages as well as helper messages
  - of course also corresponding response message types

3.3 Exception handling (Churn, connection breaks, corrupted data, . . .)

- churn is handled by the Chord algorithm
- if nodes loses the connection, then in order to rejoin the system the node would need to restart (according to current design)
  - of course the node could try to rejoin the system by connecting to the original seed node without the need to restart the JVM process, yet this is just not implemented
- the data is not persisted and stored only in-memory for simplicity sake

4 Future Work: Features you wanted to include in the implementation but couldn't finish so far

- security improvements
- key/value migration based on churn/topology changes
- communication of monitor and nodes to be done via gRPC + protobuf/thrift instead of http + JSON

5 Workload Distribution — Who did what

- since it's an individual project, all done by myself

6 Effort spent for the project (please specify individual effort)

- unfortunately I didn't keep track of time spent on the project, but I can assure you it was a lot
- I picked Scala and Akka in order to get familiar with the technologies and experiment/dive deeper into actor model system therefore a lot of time was spent on reading/learning
- the web-monitor also took a good amount to implement and figure out the deployment to heroku, trying to set up gRPC, etc.
- of course routing protocol took a lot of time to implement, as well as storage with getter/setter actors