Kadmelia implementation

Mobile and distributed systems, D7024E

Group 12

Authors:

Adrian Fingal	adrfin-1@student.ltu.se
Oscar Johansson	osajoh-9@student.ltu.se
Hugo Hedlund	hughed-0@student.ltu.se

Github:

https://github.com/Hedlund01/d7024e

LULEÅ UNIVERSITY OF TECHNOLOGY

Department of Computer Science, Electrical and Space Engineering

October 13, 2025

Contents

1 Frameworks **1**

2 Sprints **2**

2.1 Organizing Sprints 2
2.2 Sprint 1 2
2.2.1 Pinging, Joining and Finding Nodes 2
2.2.2 Store and Find Value 3
2.2.3 Unit Testing 3
2.2.4 Concurrency and Thread Safety 3
2.2.5 Command Line Interface 3
2.2.6 Review 3
2.3 Sprint 2 3
2.3.1 Planned work 3
2.3.2 Unit Testing 4
2.3.3 Concurrency and Thread Safety 4
2.3.4 Command Line Interface 4
2.3.5 Various Bug Fixes 5
2.3.6 Review 5

3 System Architecture **6**

3.1 Implementation Overview 6
3.2 Implementation Details 6
3.3 System Architecture Diagram 6

1 Frameworks

In this section, we will discuss the various frameworks and technologies that are being utilized in the project. This includes both the backend and frontend technologies, as well as any other tools that are essential for the development process.

Programming language: Go is used as the primary programming language for the backend services, leveraging its concurrency features and performance.

Logging: Logrus is used for structured logging in the application, providing a simple API for logging that allows to pass the container ip address for each log entry to easily identify which node the log entry originated from.

Containerization: Docker in swarm mode is used for container orchestration, enabling the management of multiple containers.

Automation: A Makefile exsists to automate common tasks such as building and deploying the application, making it easier to manage the deployment and development process.

Testing: The built-in testing package in Go is used for unit testing, ensuring that the code is reliable and functions as expected.

Command Line Interface: The Cobra package is used to create a command line interface for the application, allowing users to interact with the application through the terminal.

Code Coverage: Go tool cover is used to measure code coverage, ensuring that the code is adequately tested and that critical paths are covered by tests (see Makefile, coverage).

Continuous Integration: GitHub Actions is used for continuous integration, automating the testing process.

Version Control: Git and GitHub are used for version control, enabling collaboration and code management.

Project Management: GitHub Projects is used for project management, providing a visual way to track progress and manage tasks.

Documentation: LaTeX is used for documentation, providing a professional and structured way to present information.

2 Sprints

2.1 Organizing Sprints

To structure and organize the work efficiently, GitHub Projects is used to create a storyboard for both sprints. The storyboard is available under the Projects tab in the repository, providing an overview of the workflow throughout the course.

2.2 Sprint 1

2.2.1 Pinging, Joining and Finding Nodes

During sprint 1 we focused on completing work on the M* tickets we had planned out in sprint 0. We started of with getting the PING procedure to work. In the process we established how each node is going to perform non-blocking I/O operations. This was done through a main go routine that reads incoming communication and spawns of new go routines to handle the incoming messsages. We also made sure that these communications have to include a message ID, so that we can match responses to requests. This main go routine is also responsible for adding new nodes to the routing table through the already provided routing table implementation.
To test what we had built so far, we had to create a mock network. This was done mostly according to the tutorial provided with the lab instructions. Once we were satisfied with the PING procedure, we moved on to implementing the iterative node lookup procedure. In order for node joining to work. This required a lot of tinkering, especially to make sure that the procudure worked as described in the Kademlia paper.
It had to be capable of sending out multiple requests in parallel, and also handle the responses. All while updating a common shortlist of nodes to probe. In the end we created a separate package for the shortnode list and made sure that it would be thread safe. Thereby simplifying the implementation of the iterative node lookup procedure.
The iterative node lookup procedure itself uses a sort of producer-consumer pattern. Where we always have at most *alpha* go routines available to send out requests. Which are fed their nodes to probe from a channel on the main procedure thread. Which in turn come from the shortlist. As such the main procedure thread is the producer, whilst the go routines are the consumers.

2.2.2 Store and Find Value

After completing the iterative node lookup procedure, we moved on to implementing the STORE and FIND VALUE procedures. As well as their coresponding iterative procedures. A lot of this work could be reused from the iterative node lookup procedure. With only some data structure additions for handling the storing and finding of values on nodes.

2.2.3 Unit Testing

To ensure that we were on the right track during development we created unit tests for almost all of the implemented functionality. Like creating a mock network, as well as writing specific test for the procedures.

2.2.4 Concurrency and Thread Safety

As mentioned earlier, we focused on concurrency and thread safety from the begging. Which has made our diagnosing of issues a lot easier. We use mutexes for any mutually shared data structures and create separate go routines for handling concurrent tasks.

2.2.5 Command Line Interface

Finally we tried to implement simple command line interfaces for the nodes.

2.2.6 Review

To sum up sprint 1, we implemented the M* tickets we had planned out in sprint 0. We focused on concurrency and thread safety from the start. Implemented unit tests to ensure we had sufficient coverage. All of the necessary procedure s were implemented and tested. Finally we also implemented a simple command line interface for the nodes.

2.3 Sprint 2

2.3.1 Planned work

Under sprint 2 we plan to complete as many U* tickets as we can manage. Which of the tickes we will be able to complete we cannot say ahead of time. But we plan to discuss which ones we would like to complete the most. Such that we arrive at some order of priority for the tickets.

2.3.2 Unit Testing

In sprint 2 we contiued to create unit tests to meet the required code coverage of 50%. Altough we create multiple unit test from the begining of the project in sprint 1. They were mostly for our own created code, so the whole project did not have sufficient code coverage. In sprint 2 we created unit tests for the provided code, as well as for our own code that was not covered in sprint 1 to meet the total 50% code coverage requirement.
The mindset of creating unit tests helped us create and understand several hard concepts in the Kademlia protocol. For example, using TDD (Test Driven Development) to create the unit test first and then implement the functionality to make the test pass, this forced us to think about the functionality in a different way. This helped us understand the Kademlia protocol better and also helped us create a more robust implementation.

2.3.3 Concurrency and Thread Safety

During this sprint we continued to focus on concurrency and thread safty during the implementation tickets. This included using mutexes for any shared data structures and creating separate go routines for handling concurrent tasks. Because of our focus on concurrency and thread safety from the start of the project, we were able to avoid many potential issues and ensure that our implementation was robust and reliable. This can be does not fail because of race conditions via the go race flag. Altough this race flag is not the best way to measure all race conditions, it is still a good indicator that we have done a good job with concurrency and thread safety.

2.3.4 Command Line Interface

In sprint 1 we started to lay the foundation for command structure, but due to the needed interactivity in the requirements, we rewrote the command line interface to be more interactive. This was done using the *Cobra* package, and hijacking the main thread for the interactive console, while letting the other go routines handle the network communication and other tasks. This way we could have a interactive console, while still being able to handle incoming messages and other tasks in the background. This interactive console allows us to run commands like *put* and *get* to sotre and retrive values from the Kademlia network via the command line interface of each docker instance of the node.

2.3.5 Various Bug Fixes

We encountered several bugs, and wrong implementation of the kademlia protocol during sprint 2. This included bugs in the iterative lookup procedure, as well as joining procedure. This was addressed by going back to the Kademlia paper and the SourceForge implementations details and re-reading the relevant sections. This helped us understand the protocol better and also helped us identify and fix the bugs and wrong assumptions in our implementation.

2.3.6 Review

Our stated plan for sprint 2 was to complete the remaining M tickets, as well as as many U tickets as possible. While this was the plan, due to time constraints related to other courses, mainly the project course, we decided to proritize completing the M tickets and putting less focus on the U tickets. This ment that we were not able to complete any U tickets. Even if this was not our original plan, we are still satisfied with the work that we have achieved in both sprint 2 and the course as a whole.

3 System Architecture

3.1 Implementation Overview

While implementing the Kademlia DHT, our focus was to create a modular and extensible system, that can easily be maintained and expanded upon in the future. To achieve this, we sought to seperate out as many components as possible, keeping each file and package focused on a single responsibility, while also utilizing handlers to abstract away the underlying implementation details of each component. This approach allows us to easily swap out components, such as the network layer, without affecting the rest of the system.

3.2 Implementation Details

One important aspect of our implementation is how we handle docker swarm mode to spin up multiple instances of our Kademlia node and form a network. Each node needs to know the IP address and the ID of a node to join the network via that node. To facilitate this, we chose to use a bootstrap node, which is a already known node in the network that all subsequent nodes can connect to. This bootstrap node is started first, and has a hard coded ID in the docker compose file, but can easily be exchanged for any other ID by changing the docker compose fil to use a environment variable instead. All other nodes are then started and knows the docker service name of the bootstrap node and gets its ID via a CLI flag when started. This way, all nodes can connect to the bootstrap node and join the network.

3.3 System Architecture Diagram

A high level system architecture diagram is shown in Figure 1. The diagram illustrates the main components of the Kademlia system and some of their interactions. Note, this is a simplified view of the system and does not include all components and interactions. The main components are described below:

- **Network Layer:** This layer handles all network communications, including sending and receiving messages between nodes. It abstracts away the underlying network implementation, allowing for easy replacement or modification. As of now, there exists a mock network implementation for testing purposes as well as a UDP based implementation for actual network communication.
- **Kademlia Node:** This component represents a single node in the Kademlia

network. It manages its routing table, handles incoming requests, and initiates outgoing requests to other nodes.

- **Routing Table:** The routing table is responsible for maintaining a list of known nodes in the network. It organizes nodes into buckets based on their distance from the current node, facilitating efficient lookups and routing. This implementation does not implement the correct binary tree structure as described in the Kademlia paper, but rather uses a simpler list based approach for storing nodes in each bucket (given by the lab instructions).
- **Message Handlers:** These components are responsible for processing incoming messages and executing the appropriate actions based on the message type (e.g., PING, STORE, FIND VALUE). There also exsists temporary message handlers which are used for waiting for responses to outgoing requests, such as waiting for a PONG response after sending a PING request. These temporary handlers are removed once the response is received or a timeout occurs. But they are removed from the diagram for simplicity.
- **Command Line Interface (CLI):** The CLI provides an interactive interface for users to interact with the Kademlia node, allowing them to execute commands such as joining the network, storing values, and retrieving values.

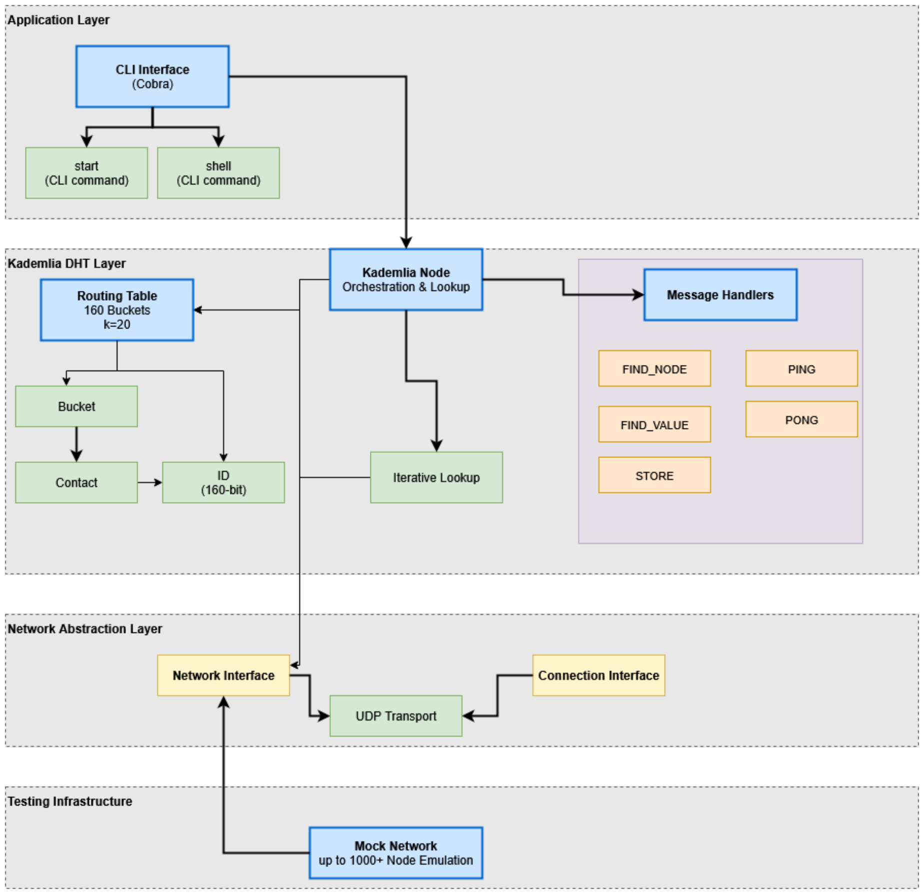

Figure 1: Component diagram of the Kademlia DHT system showing the layered architecture and component relationships.