Assignment 2: Distributed Hash Table and the Chord protocol

INF-3200 Distributed Systems Fundamentals

This project will provide you with hands-on experience on designing and implementing a distributed system. You will implement a distributed key-value store in the form of a distributed hash table (DHT), and you will investigate the characteristics of your design.

A hash table is a fundamental data structure for implementing associative arrays, such as Python's dictionaries. Key-value pairs are stored in an array by using a deterministic hash function to map each key into an integer that is then used as an index into the array. This allows look-ups by key in constant time, O(1).

A distributed hash table is a hash table where the storage is spread across many computers in a network, typically an overlay network atop an existing network such as the Internet. Compared to a single machine system, distributing the data across multiple machines may improve performance, increase scalability and/or provide better fault-tolerance. These benefits come with the cost of higher complexity and a larger development effort.

A distributed hash table needs some scheme for partitioning the key-value pairs among the many nodes in the overlay network, and a scheme for routing requests to the node that holds a requested key. Chord [1] is a distributed hash table and protocol that works by mapping both nodes and keys onto an "identifier circle" via consistent hashing. Chord will be the basis for the system you build for this assignment. The Chord protocol is described in the paper found in the references of this document, with additional explanation in the textbook [2], on pages 91 and 333 (4th edition).

This is the first of two assignments; completing both assignments is mandatory in order to qualify for the exam in this course. Please read the Requirements section carefully.

Groups

This assignment can be completed in groups of one to three members. It is recommended that you stick to the same group throughout the semester.

When working in groups, it is important that all group members contribute adequately to the to the final hand-in. Taking credit for other people's work, for example by not contributing to the group work, can in the worst case be considered plagiarism or cheating.

If you are working in a group, all group members must hand in their work separately on Canvas, even if the code and report is identical. **Every student is responsible for handing in their own work in time, even in group work**.

See Canvas for deadline.

Requirements

The requirements are absolute - be sure that your work satisfies all of the formal requirements before handing in the assignment.

Report

- The report should be approximately 1400 words long.
- The report should explain the protocol that has been implemented, its advantages and disadvantages, and explain how it was implemented.
- If something in the implementation is not working as expected or intended, it should be explained in the report.
- The report should be handed in as a PDF (and preferably typeset with LaTeX). See "Writing your report" below.
- The report should contain at least one experiment, measuring the throughput of your system. See the experiment section below.
- The results of your experiment should be presented with a plot in vector graphics (.pdf or .svg being the most common formats), with error bars showing standard deviation.
- The report should provide citations for work that you reference. For this assignment, it would for example be natural to cite the Chord paper [1], in addition to other sources that you might find useful. References should be in the IEEE citation style.
- Your report should be structured like a scientific paper. Use the Chord paper as an example.
- The report should preferably be typeset in LaTeX, but this is not a requirement. To get started with LaTeX, if you do not have a local installation, you can try Overleaf[1], which provides an online environment.

[1] overleaf.com

Code

- Follow the API specification for this assignment. See the API section below. **Implementing the appropriate API is very important, and your assignment can be rejected if it does not satisfy the API.** This requirement is absolute and programming language-agnostic.
- Your implementation should support PUT and GET operations - the system should maintain the key/value mapping of each insert, and thus retrieve the correct value on GET operations.
- Data that is PUT into the system should be placed on its corresponding node using consistent hashing, as described in the Chord paper.
- Your implementation should include appropriate use of finger table implementation so that the complexity of look-ups is better than a linear search.
- Your code **does not** need to support the dynamic leaving and joining of nodes. It can be launched with a fixed amount of nodes - see "Launching your system" below.
- No fully-connected networks are allowed. Nodes should only be aware of other nodes if they are either their neighbors, or entries in their finger table. An exception to this is immediately following initializing the system - see "Launching your system" below.
- Support running the service with at least 16 nodes.
- Any (arbitrary) node in the network should be able to serve incoming requests from a client, i.e. you need to forward GET and PUT requests to the node responsible for storing the data according to the hashing algorithm. Your implementation **does not** need to support the handling of multiple requests in parallel, but you are free to implement and test this if you want.
- Store the key/value data in memory only, and do not persist it to disk.
- Communication between nodes should only be done through network protocols. Communication through the cluster file system is not allowed.
- Your code must be able to run on the cluster.
- Your code should be launched from the cluster using the run.sh script as denoted in the previous assignment. This script should handle any setup and installation required to run your distributed hash table.

Structure

- The assignment should be handed in contained in a zipped folder (.zip), named after your UiT/Feide username (e.g. aov014). This root folder should contain a folder with the same name. This folder should contain

two folders: "doc", containing your report (and supplementary material if any), and "src", containing your implementation.

Example structure:

```
aov014.zip/
    aov014.zip
    README.txt
    src/
        code.py
    doc/
        report.pdf
```

Experiment

Your report should contain the results from at least one experiment, where you measure the throughput of your system in operations (PUTs *and* GETs) per second.

The experiment must be performed for the following network sizes: 1, 2, 4, 8, and 16 nodes. You may also include additional results for other sizes, but those listed are required. Do at least 3 runs of each trial, and plot the results in vector graphics in your report, with error bars indicating standard deviation.

An example plot is shown below. It is recommended to use a library for scripting your plots, for example `matplotlib`[2] in Python.

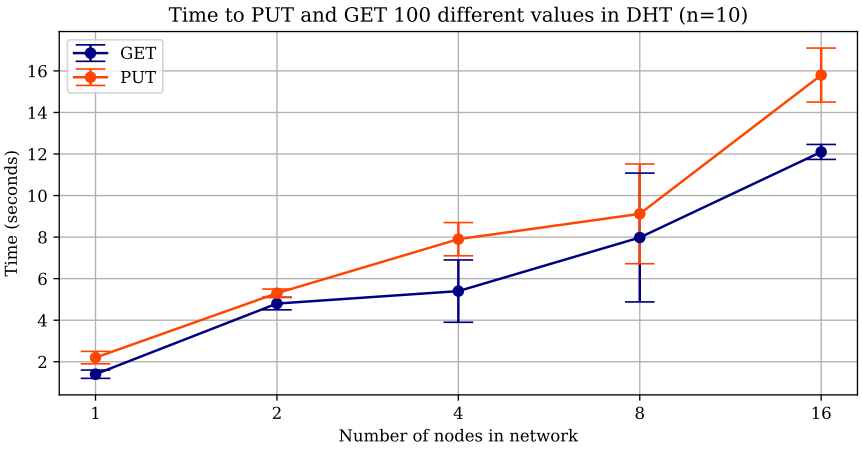

Figure 1: An example plot. Note that the image is in a vector format (.pdf) so it does not get blurry if you zoom in very close to the document. This example shows average time of operations, rather than throughput.

[2] matplotlib.org

Programming

Starter code

You will be provided with a client that can be used to test functionality and the correctness of the API. Note that the client in its current state does not test the correctness of the Chord protocol - you need to implement this test code yourself.

Launching your system

This assignment does not require you to implement the dynamic membership (i.e. leaving and joining) parts of the Chord protocol. The requirements for this assignment involves insertion and retrieval of key/value pairs, and the correct traversal of the ring using finger tables, which means that the ring creation described in the Chord paper does not need to be rigidly followed. Instead, you may initialize each node with a fixed list of other nodes in the system (i.e. IP/port pairs), and use this to organize your consistent hash ring. **However, only addresses of neighbors and finger table entries may be used for the rest of the protocol.** This is to avoid having a fully connected network. In other words, you may initialize your system with a list of all other nodes, but the list must be dropped immediately after organizing your ring and finger table. It may not be used for handling requests.

API

Nodes should implement a HTTP server that can handle the following requests:

```
HTTP GET
 - url: /storage/<key>
        Returns HTTP code 200, with value, if <key> exists in the DHT.
        Returns HTTP code 404, if <key> does not exist in the DHT.
 - url: /network
        Returns HTTP code 200, with list of known nodes, as JSON.

HTTP PUT
 - url: /storage/<key>, body: <value>
        Returns HTTP code 200. Assumed that <value> is persisted
```

The API can be checked by the test-client that you are provided with. If your code does not satisfy the API, we will be unable to test it. As mentioned in the requirements, **it is very important that your implementation satisfies the API**.

Hints and other notes

- You share the cluster with other students and users, so be respectful and try to keep your resource consumption to a minimum.
- Port numbers are global for all users on a node. It is easy to collide with other users when using common port numbers - you should probably choose port numbers in the ephemeral port range from 49152 to 65535.
- Remember that the starter code does not necessarily test the correctness of the protocol. It is up to you to evaluate this.
- To avoid excessive resource usage, you should program a reasonable time-to-live for your processes, so that they terminate automatically after a given time.
- The text for the previous assignment includes notes on how to use the cluster.
- If something doesn't work, make sure to document it and explain it in the report.
- Finally, once again, remember to follow the API. Stick to the starter code and test your implementation with the starter code client if you are worried about this requirement.

References

[1] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. 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.

[2] M. v. Steen and A. S. Tanenbaum, *Distributed Systems*. Maarten van Steen, 2023.