Mandatory Assignment 1: A Distributed Key-Value Store

UiT INF-3200, Fall 2021. Due date: Tuesday September 28 at 12:15

Mike Murphy Otto Anshus

Introduction

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.

Background

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

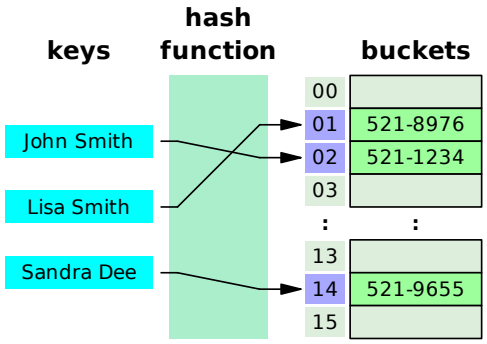

Figure 1: Example hash table (Wikipedia)

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

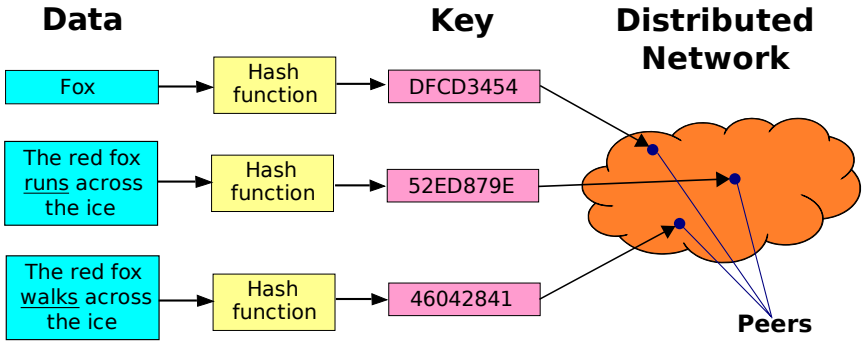

Figure 2: Distributed hash table (Wikipedia)

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 amongst 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 that works by mapping both nodes and keys onto an "identifier circle" via cryptographic hashing.

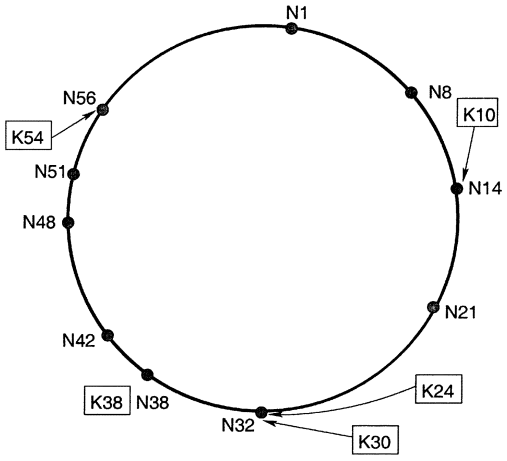

Figure 3: Chord identifier ring with ten nodes and five keys [1]

Chord will be the basis for the system you build for this assignment.

For more information on Chord, refer to Mike's lecture on the assignment (notes on Canvas), the summary of Chord in your textbook [2, pp. 83–84], and especially the original Chord paper [1].

Assignment

Your assignment is to build a distributed hash table based on Chord [1]. You will create a system of storage nodes that act as a single distributed key-value store. They will accept HTTP PUT and GET requests from a client to store and retrieve key-value pairs.

The storage nodes will run on the development cluster (`uvcluster`). You should receive an email from IFI Kontoadmin with your cluster login information soon (if you have not already). If you do not receive a password, talk to one of the TAs and we will get it sorted out.

There is also a separate document on Canvas with more information about using the cluster.

Your hand-in will consist of source code for your key-value store and a report describing your architecture, design, and implementation. You will also be required to give a demo, where a TA-controlled client will send requests to your data store and report statistics.

Groups of Two

The assignment should be done in pairs. It is allowed to work alone, but we **strongly recommend that you work in groups**. It has been our experience that students who work alone have more trouble with the assignments, especially during these times of mostly-online classes. Working in a group gives you someone to bounce ideas off of when you get stuck. Please only work alone if you have a very strong preference or circumstances that compel you to do so.

It is also allowed to work in a group of three, but be warned that we will expect more thorough work from the groups of three.

Please choose your assignment groups from within your same colloquium group, so that you can work together in the colloquiums. Keep in mind that these colloquium groups are separate for social distancing reasons. We need to minimize cross-contact.

If you are having trouble finding a partner, talk to the TAs. We will try to connect you to someone else who is also looking.

Basic Architecture

Client

A client application issues PUT and GET requests to the storage service. Your storage nodes should implement the API specified below, redirecting incoming PUT and GET requests to the appropriate node.

We have provided a sample client and a dummy node to get you started. The client does some randomized PUT and GET operations, and the dummy node implements the facade of the API with no real internal logic. Your job is to fill-in this internal logic (you may also completely re-implement it in the language of your choice). You may add additional API calls for inspection and debugging if you like. However, it is important that you do not change the core API

behavior specified here. This is so that your API remains compatible with the TA-controlled client that we will use during the demos.

Storage nodes

Upon receiving a request from a client, the storage node is responsible for completing the requested operation. This likely involves contacting other nodes to handle the request. The client should be able to send a request to any storage node and still receive the requested key.

API

Your storage nodes should implement the following HTTP API calls:

- **PUT /storage/*key***: Store the value (message body) at the specific key (last part of the URI). PUT requests issued with existing keys should overwrite the stored data.
- **GET /storage/*key***: Retrieve the value at the specific key (last part of the URI). The response body should then contain the value for that key.
- **GET /neighbors**: Retrieve a JSON array of other storage nodes that this node is connected to. Each node should be in *hostname*:*port* or *IP*:*port* format. Example response body:

  ```
  [ "compute1-1:8080", "compute1-2:8086", "compute2-4:10000" ]
  ```

Requirements

Hand-in

Your delivery must include:

1. **Source code** for your DHT, with instructions on how to run it
2. **Report**, including results of required experiments

You should zip up (or tarball) your source code and report, and upload them to Canvas before the due date: Tuesday September 28 at 12:15 (start of colloquium period).

Then, in the colloquium period, you will give a short **demo** where you describe and run your solution.

Implementation / Source Code

- Create a distributed key-value store that partitions the key space among nodes in a consistent and structured way. We recommend implementing the Chord [1] protocol. If you would like to implement something else, talk to the TAs first.
- No completely-connected networks. Each node must be aware of only a subset of the other nodes, and this should be reflected in the response of the `GET /neighbors` API call.

- Support running the service with at least 16 nodes. We require you to run approximately 16 nodes at the demo session. Note that we do not require you to support nodes joining and leaving when serving storage requests.
- Any (random) node in the network should be able to serve incoming requests from the client, i.e. you need to forward GET and PUT requests to the node responsible for storing the data according to the hashing algorithm.
- Store the data in-memory (please avoid storing any data on disk). It is not necessary to persist data between separate runs of the system.
- Your source code should include a README file with instructions for running your solution on the cluster.

Required Experiment

You must run an experiment to measure the throughput of your system in PUTs+GETs per second at the following sizes: 1, 2, 4, 8, and 16 nodes. You may also include additional results from other sizes, but those listed are required.

Do at least three runs of each trial.

Plot your results in your report, including error bars for the standard deviations.

Report

Your report should describe your work in the format of a scientific paper.

- Your report must contain a plot of the results from the required experiment. Other experiments and results are encouraged.
- Your report should render plots and diagrams in a vector-graphics format, so that they do not become fuzzy or pixelated when zoomed in.
- Your report should be approximately 1200 words long.
- Your report should be preferably typeset with LaTeX.
- Your report should give citations for other work that you reference, including other work that inspires your system or that you compare your system to. In this case, you should at the very least cite the Chord paper [1] as we have here.
- Avoid citing Wikipedia. Drill down and try to find the primary paper that was the origin of the concept.
- Your report should be structured like a scientific paper. Use the Chord paper as an example, and see the "How to Write a Report" document on Canvas for more tips.

Demo

The demo will be an informal presentation of your solution. There is no need to make slides, but you should talk briefly about your architecture, design, and

implementation. Especially talk about ways you think your solution may have differed from other groups'.

You will also start your system running on the cluster. The TAs will then run a client to exercise your network and collect statistics. The TA client will expand on the hand-out client with additional tests and maybe some surprises.

Other Things to Keep in Mind

- You share the cluster with the others students, so please try to keep resource consumption to a minimum. It is especially easy to clash with port numbers. Avoid common port numbers like 80 or 8000 and choose something obscure from the ephemeral port range: 49152 to 65535.
- You should also add a reasonable time-to-live for each process so that it terminates after a given amount of time if you forget to shut it down it yourself.
- Scripting is your friend. You will make things easier for yourself if you can script the startup, evaluation, and shutdown of your network so that you can run those things with simple commands rather than having to start and stop each node by hand.
- The starter code we gave you may be written in Python but you are not limited to Python. You are allowed to implement your solution in any language that you can get running on the cluster, as long as you implement the given HTTP API. If you want to use a language that you are more familiar with, that is fine. If you want to challenge yourself with an unfamiliar language, that is great!
- The second assignment will build on this one. So don't leave yourself with an unmaintainable mess of source code.
- Start early, fail early. (Make it better early.)
- **Deadline: Tue Sept 29, 2020 at 12:15 (start of colloquium period).** Your hand-in should be uploaded to Canvas by the beginning of the colloquium period and you should be ready to give your demo during the colloquium.

References

[1] I. Stoica *et al.*, "Chord: A scalable peer-to-peer lookup protocol for internet applications," *IEEE/ACM Trans. Netw.*, vol. 11, no. 1, pp. 17–32, Feb. 2003, doi: 10.1109/TNET.2002.808407.

[2] M. van Steen and A. S. Tanenbaum, *Distributed systems*, 3rd ed. 2017.