INF3200: Distributed Systems Fundamentals
Mandatory Assignment 1

Tue Sept 01, 2020

Due Date: Tue Sept 29, 2020 at 12:15 (start of colloquium period)

1 Introduction

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

Contents

2 Background: Distributed Data Stores

A *distributed* data store is a network of computers providing a service for storing and retrieving data. 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 bigger developing effort. A *key-value* store is a type of distributed data store designed for storing and retrieving records identified by a unique key.

Processes across multiple machines provide a single service, passing messages between one another. The processes communicate according to the rules of the chosen architecture. Structured and unstructured *peer-to-peer* architectures are both suitable architectures for building a distributed data store. A distributed hash table is the most common way of organizing the processes in peer-to-peer systems.

The data is distributed between the nodes through a partitioning scheme. The partitioning scheme dictates which node is responsible for storing a given key-value pair (k, v). A simple solution would be to assign each pair to a random node, but then each lookup would have to consult every note to see if it has the key. In this case, the store operation is $O(1)$, but lookups are $O(n)$ for a store with n nodes.

A more refined solution is to partition the key space and to assign a node to each partition (e.g. divide the keys alphabetically and assign keys that start with a–m to node n_1 and keys with n–z to n_2). This adds a consistency to the lookup operation: given a key, we know exactly which node should store that key. Now both storage and lookup are $O(1)$. However, it's likely that more keys start with some letters than others, so they load may not be balanced across nodes.

A further refinement then is to use *consistent hashing*. Cryptographic hash functions such as MD5 or SHA1 are designed to give a uniform distribution of resulting hashes. We can take advantage of this property to help distribute keys evenly across nodes in our system. Instead of partitioning the key space by the key itself, we can hash the key first and partition the key space based on the hash instead. This maintains the $O(1)$ storage and lookup, but also spreads the keys evenly to the nodes in the system. This was the main innovation of the Chord protocol [**stoica2003chord**] (there is also a nice introduction to Chord in your textbook [**Steen2017**]).

The other innovation of Chord is a way of scaling with large numbers of storage nodes. To achieve the $O(1)$ storage and lookup we noted before, every storage node must have a list of all other nodes and their assigned key-space partition. As the number of nodes grows, it becomes more difficult to keep this list of nodes in sync. Adding a node and updating this table is an $O(n)$ operation. Chord arranges nodes logically in a ring, with each node connected to $\log_2 n$ other nodes. With Chord, storage, lookup, and membership changes are all $O(\log_2 n)$.

Chord is the basis of Amazon's Dynamo store [**decandia2007dynamo**], which in turn is the basis of Apache Cassandra [**lakshman2010cassandra**].

Chord will be the basis for the key-value store you build for this assignment.

3 Assignment

Your assignment is to build a distributed key-value store with consistent hashing, along the lines of the Chord protocol [**stoica2003chord**]. You will create a system of storage nodes that act as a single distributed data 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 have received an email from IFI Kontoadmin with your cluster login information. If you have not, talk to one of the TAs and we will get you 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.

3.1 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**. Last semester when the university went all digital mid-semester, many students decided to work alone, and we noticed that those students were the ones who had the most trouble with the assignment. 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 push you to work alone.

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 and 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.

3.2 Basic Architecture

3.2.1 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, so that your API remains compatible with the TA-controlled client that we will use during the demos.

3.2.2 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.

3.3 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* format.

Example response body:

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

4 Requirements

Hand-in

The delivery must include:

1. **Source code** (programming language of your choice[1]) with instructions on how to run.
2. **Report**

You should zip up (or tarball) your source code and report, and upload them to Canvas before the due date: Tue Sept 29, 2020 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.

4.1 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 [**stoica2003chord**] protocol. If you would like to implement something else, talk to the TAs first.
- No complete 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 a minimum 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, 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.

4.2 Report

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

[1] and supported by uvcluster.cs.uit.no

- Your report must contain a graph showing the throughput as transactions per second (y-axis) of the system with 1, 2, 4, 8 and 16 nodes (x-axis) for both PUT and GET.

  Be careful to render your graph in a vector-graphics format, so that it does not get fuzzy 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 [**stoica2003chord**] 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. A typical computer science paper includes the following sections:

  Introduction A brief overview of your solution.

  Background A brief review of concepts necessary to understand your solution (in this case, key-value stores, distributed stores, and distributed hash tables).

  Related Work A review of other papers related to your work (in this case, Chord), with notes on how your work differs.

  Description of System (Architecture / Design / Implementation) Actually describe your work. Start with a high-level description of the fundamental *architecture* of your solution. Then describe other *design* decisions that were made in the process of fleshing out the architecture. Finally, describe the low-level *implementation* details such as hardware and programming language.

  Evaluation Describe the methods used to evaluate the system. Define the metrics used to measure performance, and the procedures used to gather metrics.

  Results Give the results of the evaluation, including graphs.

  Discussion Discuss your solution and your experiment results in a more qualitative way. What are the positives and negatives? Discuss unexpected results and their implications.

  Future Work Discuss possible ways to improve or expand the system.

  Conclusion Briefly restate the key points of your work and wrap up the paper.

  Since the parameters of this assignment are relatively narrow, you may not have a lot of content for every section listed. You may shorten or combine sections, but try to follow the template.

  Also see the papers cited here for examples, especially the Chord paper.

4.3 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 run your system on the cluster, and the TAs will 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.

5 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.