INF3200: Distributed Systems Fundamentals
Fall 2019
Mandatory Assignment 2: Membership

Mike Murphy Aakash Sharma

Hand-out: Tuesday October 15th, 2019
Due date: Tuesday November 5th, 2019

Introduction

This assignment will build on the key-value store network you built for Assignment 1. This time, your task is to allow nodes to join and leave the network dynamically.

As before, we will drive your network with our own client code during the demo. So be sure to implement the API as described here (appendix A), and do not change relationship between client and network for your implementation. However, we do encourage you to add your own testing and debugging functionality.

Mike will be the main TA to contact for this assignment. Email him at `michael.j.murphy@uit.no` with any questions, or stop by his office, Realfagbygget A249.

1 Overlay Network

As before, your network should be an overlay network of nodes on our compute cluster. The overlay network should be based on Chord [1], where nodes are ordered by a hash of their ID, and each node has a pointer to its successor node (and optionally others).

Your nodes will also implement additional HTTP API calls to allow them to join and leave a network, and also to simulate a crash. See appendix A for API details.

Your network should still store and retrieve key-value pairs, but you are not required to move existing pairs as the network changes. That is beyond the scope of the assignment.

You are not required to implement Chord's finger tables, but we encourage you to try. Whether you implement them or not, spend some time thinking about the trade-offs of having or not having finger tables as the network changes. We expect to see some discussion of this in your report.

2 Evaluating Your Network Implementation

You should perform experiments to measure the following metrics for your network:

Time to grow network to 50 nodes Start with 50 running nodes, each in a single-node state, then use join API calls to join all 50 into a network. Measure the time from issuing the first join call to reaching a stable 50-node network. API calls may issued sequentially, or in a burst.

Time to shrink network from 50 nodes to 25 Start with 50 nodes in a stable network, then use leave API calls nodes leave the network. Measure the time from issuing the first leave call to reaching a stable 25-node network. API calls may issued sequentially, or in a burst.

Network tolerance to bursts of node crashes Start with 50 nodes in a stable network, then use the simulate-crash API call to "crash" a node. The "crashed" node should then stop responding to all internal network messages. After some time-out, the network should notice that the node is unresponsive and then route around it, becoming a stable network of 49 nodes.

Now repeat this procedure with a burst of two simulated crashes, then three, and so on. How large of a burst of crashes can your network tolerate by still returning to a stable state?

Designing these experiments is part of the assignment. You will have to write code to probe your network and determine whether it is stable. You will also have to decide what timer mechanism to use and exactly when to start and stop it. Be sure to describe your methodology in the report.

3 Requirements

The delivery must include source code and report. You will also have to participate in a demo session similar to the first assignment.

3.1 Source Code

- must run on the cluster `uvcluster.cs.uit.no`.

- may be in any language that runs on the cluster.
- must include a README file with instructions for running the code on the cluster (starting and stopping nodes, etc.).

3.2 Report

- must include details about your design and implementation, especially the decisions you made while implementing it. Focus on what might make your implementation different from others.
- must include the results of the required experiments described in section 2, and a description of your methodology, how you performed them.
- should be structured like a scientific paper (see appendix B).
- should be approximately 1200 words long.

3.3 Demo Session

As you saw with the previous assignment, the demo session is relatively informal. We will ask you to describe your implementation, focusing on what might set it apart from other implementations. Then we will ask you to start a network on the cluster (up to 50 nodes), and we will run our own client code to drive and test your network. So be sure to implement the API as specified in appendix A.

3.4 Other requirements

Share the cluster You share the cluster with multiple students, so please try to keep resource consumption to a minimum.

- Add a reasonable time-to-live for each process so that it terminates on its own if you forget to kill it.
- Store test data in memory. Persistence is not part of this assignment. Don't bother writing nonsense data to disk on the cluster.
- Use the ephemeral port range, 49152 to 65535.

Deadline: Tuesday November 5th, 2019

4 Notes and Tips

Use your preferred language You can write your code in any language that you can get running on the cluster. The starter code is in Python, but it is only an example. Follow your heart.

Try to think in terms of parallelism Your nodes are running as separate processes. There are race conditions waiting to occur as they update their pointers. Think about the implications of this. What kind of locking mechanisms or multi-phase join/leave process could help keep the network stable? Does the presence of finger tables contribute to fault tolerance? Be sure to discuss these things in your report.

Write helper scripts Remember: repetitive and tedious things are what computers do best. Write scripts to help yourself quickly start up, tear down, and experiment with your networks.

Don't panic Remember: we are not expecting your code to be bulletproof. Distributed systems are not easy, and we want you to be learning about the problems, solutions, and trade-offs involved.

The demo, especially, is supposed to really push your code and try to find edge cases and breaking points. This is meant to be stress test for your *code*, not for *you*. We are looking for interesting behavior, so we can all learn from it.

Mike and Aakash are here to help Talk to them if you are having trouble. (For this assignment, come to Mike first.)

Start early, fail early. (Make it better early.)

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 (TON), vol. 11, no. 1, 2003, pp. 17–32. [Online]. Available: https://doi.org/10.1109/TNET.2002.808407

A API Specification

Each node must implement the following HTTP API calls.

A.1 From Assignment 1

PUT /storage/<key> Store the value (message body) at the specific key (last part of the URI). PUT requests issued with existing keys must overwrite the stored data. Values should be stored in memory. Please avoid storing data on the disk.

GET /storage/<key> Retrieve the value at the specific key (last part of the URI). The response body must then contain the value for that key.

A2 REMINDER It is not required to move data as the network changes. So it is ok if, after joins and leaves, the network can no longer reliably find stored keys. However, if the network is stable, then new PUTs and GETs should work as expected from any node.

A.2 New for Assignment 2

GET /node-info List node key, state, and neighbors.

A2 UPDATE This is a superset of the info in the */neighbours* call from Assignment 1. It should be a JSON object as described in appendix A.3.

POST /join?nprime=<HOST:PORT> Join a network. The node must contact the neighbor node *nprime* and join *nprime*'s network.

POST /leave Leave the network. The node must go back to its starting single-node state, and the rest of the network should adjust accordingly. How you implement this is up to you. The node may notify its neighbors that it is leaving, or it can be the network's responsibility to detect the change.

POST /sim-crash Simulate a crash. The node must stop processing requests from other nodes, without notifying them first. Any request or normal operational messages between nodes should be either completely refused or responded to with an error code without being acted upon. The "crashed" node must respond correctly only to the */sim-recover* and */node-info* calls. The network should detect the crash and respond as if the node has left.

POST /sim-recover Simulate a crash recovery. The node must "recover" from its simulated crashed state and begin responding again to requests from other nodes. If the network has given up on the node, it should request to re-join the network via one of its previous neighbors.

A.3 Node-Info JSON Format

The */node-info* call should return a JSON object similar to the following example:

```
{
    "node_key": "35c25a8",
    "successor": "compute-1-1:8080",
    "others": [
        "compute-2-3:54000"
        ],
    "sim-crash": false
}
```

The members are as follows:

`node_key` (string): the node's hash-based key/ID. Should be a hexadecimal or integer representation of the key that determines the node's position in the key space. If all nodes' keys are put into a list and sorted, it should match a list of visiting each node from successor to successor. Or, it should for a stable network (*hint*, *hint*).

`successor` (string): the node's successor as a HOST:KEY pair.

`others` (array of strings): the node's other neighbors as a list of HOST:KEY pairs. This can include predecessor, finger table members, and any other nodes that this node is aware of for any reason.

`sim_crash` (boolean): true if the node is in a simulated-crash state due to a */sim-crash* call, otherwise false.

You can add more info to your */node-info* response for your own purposes, but those specified members must be present with the given semantics.

B Scientific Paper Structure

Scientific papers usually follow a standard structure, similar to the following. Try to structure your report similarly. Use the papers you have read in class as examples.

Scientific Paper Structure:

1. Introduction - describe your task.
2. Design - description of your architecture, communication requirements, lookup algorithms.
3. Experiments - describe your experiments with different metrics and what they attempt to evaluate. Explain choice of metrics.
4. Discussion - discuss positive and negative sides of your solution. Have you critically evaluated your solution?
5. Summary Did you find any unexpected results or behavior of the system? Try to investigate and find a possible explanation. This might require tweaking the source code or experiment parameters to try different ideas and compare results.

Do not forget to

- Make your figures clear and understandable.
- Cite references, insert captions and axes descriptions.