Distributed Hash Table: Dynamic Membership

INF-3200 Distributed Systems Fundamentals

Introduction

This assignment will build on the key-value store network you built for the previous assignment. This time, your task is to allow nodes to join and leave the network dynamically. As before, your network must be an overlay network of nodes on our compute cluster. The overlay network should be based on Chord, where nodes are ordered by a hash of their ID, and each node has a pointer to its successor node. Your nodes must also implement additional HTTP API calls to allow them to join and leave a network and to simulate a node crashing. See below for API details, and see the starter code for an API checker client.

This is the second 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 alone or in groups of two or three. It is recommended to retain groups across assignments. Groups of more than three are not allowed.

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.

Handing in

The deadline for handing in the assignment is published in the Assignments-section on Canvas. 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**.

API Specification

Each node must implement the following HTTP API calls. Use the supplied `api_check.py` script to check that your API is implemented correctly.

API from previous assignment

The PUT and GET for key-value pairs must work exactly as before:

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.

Remember: It is not required to move data as the network changes. It is acceptable if the network has trouble finding old keys after a join or a leave. However, while the network is stable, then new PUTs and GETs should work as expected from any node.

API extensions for this assignment

GET /node-info List node identifier/hash, state, and neighbors. The response must be a JSON object like this one:

```
{
    "node_hash": "35c25a8",
    "successor": "compute-1-1:8080",
    "others": [
        "compute-2-3:54000",
        "compute-4-5:55000"
        ]
}
```

The members are as follows:

`node_hash` (string): the node's hashed key/ID. Should be a hexadecimal or integer representation of the hash that determines the node's position in the key space.

`successor` (string): the node's successor as a HOST:PORT pair (IP:PORT is also acceptable).

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

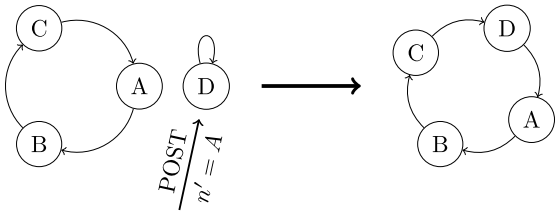
POST /join?nprime=HOST:PORT Join a network. The node that receives this message must join `nprime`'s network, as shown in Figure 1. Note the direction of this call: the POST is sent to the loner node, and the parameter indicates the network that the loner node should join.

POST /leave Leave the network. The node must go back to its starting single-node state, and the rest of the network must adjust accordingly. How you implement this is up to you. The node may notify its neighbors that it is leaving, or it could also 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 **only** to the /sim-recover call. Your network must detect that the node is not responding, and rearrange itself 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 crashed node has been excluded from the network, it should request to re-join the network via one of its previous neighbors.

You may add more info to your node-info response for your own purposes, but these specified members must be present and they must have the given semantics.



POST http://D/join?nprime=A

Figure 1: Node D joining the ring of A

Evaluating your network implementation

You must perform the following required experiments to evaluate your network:

Time to grow network to 32 nodes Start with 32 running nodes, each in a single-node state, then use join API calls to join all 32 into a network. Measure the time from issuing the first join call to reaching a stable 32-node network. API calls may issued sequentially, or in a burst. Repeat this experiment with network sizes of 2, 4, 8, 16, and 32 nodes.

Time to shrink network from 32 nodes to 16 Start with 32 nodes in a stable network, then use leave API calls to make nodes leave the network. Measure the time from issuing the first leave call to reaching a stable 16-node network. API calls may issued sequentially, or in a burst. Repeat this experiment with network sizes of 4, 8, 16, and 32 nodes, each time shrinking to half the size. So $32 \rightarrow 16$, $16 \rightarrow 8$, and so on.

Network tolerance to bursts of node crashes Start with 32 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 32 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 and still return to a stable state?

Designing these experiments is part of the assignment. You will have to write code to probe your network and determine when 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.

Repeat each trial of each experiment at least three times so that you can calculate mean and standard deviation of the measured durations. When you plot your results, plot the mean values and include the standard deviations as error bars.

Requirements

The requirements are absolute - be sure that your work satisfies all of the formal requirements before handing in the assignment. Many of the requirements are similar to the last assignment, so we will highlight the following:

1. Your code should support the dynamic leaving and joining of nodes according to the API specification.
2. Starting a node with knowledge of all other nodes in the network is not allowed for this assignment. Every node should be initialized as if member of a ring containing only itself. Nodes should then be able to join the rings of other nodes through API calls.
3. 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, *as long as the network is stable*. Data that is PUT into the system should be placed on its corresponding node using consistent hashing, but it is acceptable if the data is unavailable if its containing node leaves the network. Similarly, it is acceptable if finger tables become unstable after the network layout is changed.
4. Your hand-in must include a `run.sh` script that automates running your network with any number of nodes, as in previous assignments.

Report

- The report should be approximately 1200 words long.
- The report should explain the features that have been implemented, their purpose and how they work. Explain how they impact data location and availability, and routing through finger tables.
- 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).
- The report should contain the experiments described in the section above.
- The results of your experiments should be presented with plots 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.

Code

- Follow the API specification for this assignment. **Implementing the appropriate API is very important, and your assignment will be rejected if it does not satisfy the API.** This requirement is absolute and programming language-agnostic. You will be provided with a test script that you can use to check your API implementation.
- 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.
- Support running the service with at least 32 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 in the same manner as in previous assignments.
- You do not have to consider moving key/value pairs between nodes in case of network re-structuring in this assignment.

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/
       src/
           code.py
       doc/
           report.pdf
```

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.