Assignment 2: Distributed log server

INF-3203 Advanced Distributed Systems, February 2025

Introduction

In this assignment, you will implement a consistent distributed log server that clients can connect to in order to store their log entries. Your implementation should ensure that log entries are stored and ordered consistently between all servers.

A client will continuously connect to any server in your distributed system and post messages to it; the server is responsible for storing and distributing the data in whatever way is necessary to ensure a robust service and consistent ordering. At times, servers might exhibit characteristics similar to a system crash, such as failing to respond to messages, or halting communication. The system you design should be fault-tolerant, ensuring that it can still function even if individual nodes encounter such crashes.

To make testing and evaluation easier for you, the code that TAs will use to assess your solution will be made available to you together with the starter code. Because of this, the assignment must be solved in Python.

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 maximum three.

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.

Deadline

The deadline for handing in the assignment is in four weeks, on **March 24th 23:59**, 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**.

The problem

Imagine several clients across the world communicating with geographically distributed servers that are part of the same system, as illustrated in Figure 1.

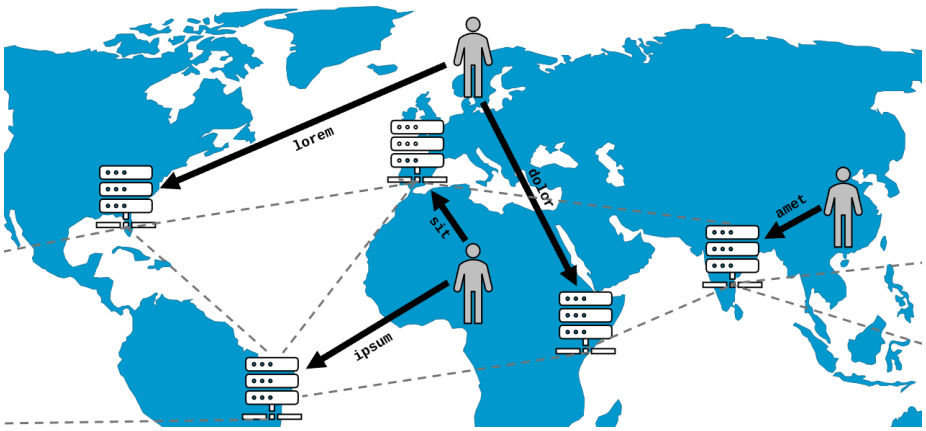

Figure 1: Users across the world entering messages of "*lorem*", "*ipsum*", "*dolor*", "*sit*" and "*amet*" to different servers comprising a global distributed system

Achieving a consistent ordering of events across these servers is an important feature of distributed systems, as inconsistent ordering can cause serious problems or abnormal behaviour depending on the application. Imagine a cash terminal or web portal of a banking system displaying different balances for customers depending on where in the world they are accessing their account from. Even worse, a user can stay in the same physical location, and be pointed to different servers by an internal load balancer when they refresh their account in their web browser. It is clear that the ability to maintain consistency between servers is an essential trait of distributed systems.

Note that the problem is not necessarily that messages should be logged in the exact same order as they were sent by the client - this is a different task involving network latency and clock synchronization. This assignment is focused on consistency between servers.

Solving the assignment

The assignment is solved by coordinating your servers so that they all produce their own individual log that is consistent with the other servers. This means that when a server receives a message from a client, it should communicate with other servers to reach consensus on how to store the message. There are several distributed consensus algorithms that are applicable to solving this problem (for example Paxos [1], Raft [2], Multi-Paxos [3] and others) and we recommend investigating their individual advantages and disadvantages when planning your implementation. A tip is that there are probably many resources available to you that make these algorithms more understandable than their original research papers are able to (although they are recommended reading for understanding how to navigate the distributed consensus landscape [2]). There are also possible

solutions outside of these common algorithms, and these are also valid ways of solving the assignment.

Crashes are simulated in this assignment, so it is important to respect the crashed state of the server as given in the pre-code. This means that a server that is in a crashed state cannot communicate with other nodes until it explicitly recovers from its crashed state by a recover request from the client.

Servers run as individual processes and should communicate with each other over the network. Using the file system for communication is not allowed; only when processes are exiting should a log be written to persistent storage.

Using a single client is adequate for this assignment.

Included files

`log-client.py` A client that will continuously send messages to any node given a list of addresses, expecting their message to be stored as entries in a distributed log. You are free to extend this file, but you do not have to. **Remember that when your hand-in is assessed, the client will be used as provided in the starter code.**

The client has the following capabilities: (1) Generate words that are sent to servers with a HTTP PUT request on a timed interval. (2) Instruct servers to occasionally exhibit crash-like behaviour for a given time-frame, using HTTP POST requests. This is the method we use for simulating faults in this assignment. (3) Write entries that were presumed successfully logged to a file on the local file system. The name of this file is given by the `output_id` parameter. The list of entries is persisted as a file for easier evaluation and debugging.

The client takes an input parameter to indicate a *Scenario*, which is a system configuration that dictates time between log messages, total number of messages, duration of crashes, and number of concurrent crashes. Increment this parameter to test the stability of your implementation.

Both the client and server takes an input parameter *output_id* which indicates the filename that output logs should be given. This is used for checking the correctness of the log server (see `log-comparer.py` below)

`log-server.py` A server that receives and stores log messages from clients. The server as provided does not contain any implementation for sharing data with other servers, so this file must be extended to accommodate this.

The server already has certain functionality, which should be left intact:

1. HTTP PUT is used to receive incoming log messages
2. POST to `host/crash` turns the server to a crashed state, and should not respond to any messages other than "recover" and "exit"-messages, until the system has recovered.

3. POST to `host/recover` turns the server to a non-crashed state
4. POST to `host/exit` writes the local log to the local file system and exits the program. You can and should modify the mechanisms of the log, but regardless of your implementation, it should be written to a file on program exit.

`log-comparer.py` A script that compares logs from a single run of the client, identified by the given *output_id*. The script checks the following conditions:

1. All messages sent by the client should be present in the logs from the server
2. All server logs should be identical to each other

If these conditions are met, it means that the servers were successful in creating a consistent distributed log. **When your hand-in is assessed, the log-comparer will be used as provided in the starter code.**

`run.sh` A script that runs a single client and a given number of servers with a common *output_id*.

Report

The report should be approximately 1200 words long and include citations for work that you reference. References should be in the IEEE citation style.

In your report, you should explain the design of your solution and how you believe it accomplishes a consistent log. Explain why you chose this solution, and what its strengths are. Explain which *scenarios* (see `log-client.py`) it was able to successfully complete, and which scenarios failed, and, most importantly, *why* this was the outcome.

Code

- You must stick to the pre-code that is implemented in Python. Use the included files to test your implementation. Solve the assignment by modifying and extending the log-server script.
- Your server implementation should provide a consistent log that passes the tests implemented in the log-comparer script, when the pre-code client is used to communicate with the server
- Respect the server's crashed state and do not implement a server that communicates with other nodes when crashed
- Keep the log-server's HTTP endpoints (PUT, POST/crash, POST/recover, POST/exit intact. Feel free to implement more endpoints and forms for network communication, but they must only respond when the server is not simulating a crash.

- The system is expected to function primarily with 5 servers and 1 client.
- The pre-code is configured to run on a single machine, using localhost and different port numbers. You can solve the assignment on the cluster if you wish, but it is not required.

Structure

- The assignment should be handed in contained in a zipped folder (**either .zip or .tar.gz)**, named after your UiT/Feide username (e.g. aov014). This root folder should contain two folders: "doc", containing your report (and supplementary material if any), and "src", containing your implementation.

Example structure:

```
aaa014.zip/
   aaa014/
       README.txt
       src/
           code.py
       doc/
           report.pdf
```

References

[1] L. Lamport, "Paxos made simple," *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pp. 51–58, 2001.

[2] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX annual technical conference (USENIX ATC 14)*, pp. 305–319, 2014.

[3] R. Van Renesse and D. Altinbuken, "Paxos made moderately complex," *ACM Computing Surveys (CSUR)*, vol. 47, no. 3, pp. 1–36, 2015.