TECHNION | The Henry and Marilyn Taub Faculty of Computer Science

Distributed Systems Course 236351

Practical Assignment

Bear-bones State Machine Replication

Roy Friedman and Yaron Hay

Fall 2024/25

Version 1.0

1 The Big Picture

In this exercise you will implement a state machine reapplication service using consensus from (almost) scratch. For this exercise you need to use the knowledge you learned in the course to provide a consensus mechanism that is simple yet effective. You are allowed to implement protocol, but you should base your implementation on Paxos and its MultiPaxos Variant.

To simplify the implementation, we instruct you to use a coordination service (etcd) for implementing important primitives such as failure detectors and group membership. However, the objective is to implement a replicated log for ordering client transactions without running them through etcd.

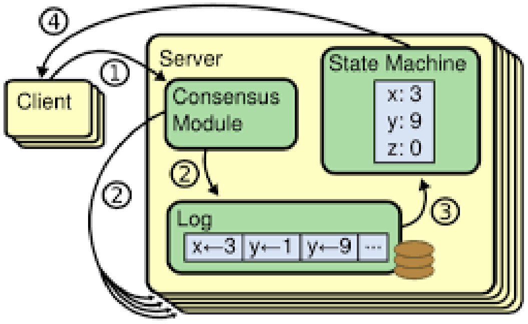

Figure 1: Illustration of State Machine Replication. Credit reserved to the Raft Paper.

2 System Components

Multi Consensus. Implement a Paxos-like protocol, recall the two phases for each round. Then extend the protocol to a Multi-Consensus protocol by running increasing instances of the consensus protocol; each instance has its own state variables and its own rounds.

Replicated Log. Use you consensus protocol to create a replicated log. You should have checkpoints that allow you to dispose of unused consensus instances when **all** nodes have learned the decision values[1].

Log Compaction. Periodically take snapshots of your state machine, so you can clear the log to a specific point. However, when clients recover from a failure or join the system they need to catch up to the current log state. Snapshots provide the state of the system after a specific log entry; therefore, nodes need to be aware that log entries prior to that entry do not exist anymore.

Membership Changes. You should support changes to that membership state: nodes may join or leave. Since this changes the acceptor set, you should adapt the the consensus protocol to support this. Use etcd to implement membership change process for tasks such as: listing pending changes and submitting them, creating barriers and locks, and node discovery (e.g. IP+Port).

3 Specifications

3.1 Recommended Stages of Implementation

Stage#	Task	New Assumptions
1	Failure detection and leader election.	Initially all nodes are alive before the system starts, and nodes do not join or leave.
2	Consensus messages for multiple instances.	
3	Replicated Log and client demo application	
4	Node recovery	Node may start at any time, can crash and recover.
5	Snapshots and consensus disposal	After a consensus instance is disposed, it is illegal to reinvoke it.
6	Membership posting membership changes to etcd	
7	Changing the acceptors set according to the membership state. Use etcd for synchronization.	

[1] Recall that you may potentially have an infinite number of rounds in a consensus instance depending on the failure pattern. However, this is an edge case.

Implement a sample application that uses you state machine replication implantation for demonstration purposes. See Appendix A for an example.

Client Communication A client may contact any server in the system, even though the specific server is not responsible handling the request. In such case, the server should forward the request to the server responsible on behalf of the client.

3.2 Client API

RESTful API Clients must contact the system with a RESTful API that is designed correctly.

Client Semantics You may implement any application that you wish as an demonstration of your implementation, or use our example in Appendix A. Creativity *may* credit you with bonus points; however, it is recommended to keep it simple, so that you do not loose focus from the project objective.

Consistency Guarantees The system must expose a RESTful API to clients allowing various functionalities with the following characteristics:

Logical Operation	**Input**	**Consistency**
Operation involving returning the state of some object (A Read Operation)	Any	Sequentially Consistent
The same operation as above	Any	Linearizable
The same operation as above	Any	Linearizable

Implement at least one for each category.

4 Guidelines

Use the techniques and principles you have learned in the lectures and tutorials, and demonstrate them in you project.

Development Language Develop your project in either Go.

Message Communication Use gRPC for any internal communication between the servers. Use a RESTful API over HTTP for any communication between clients and servers.

Coordination Use etcd for implementing failure detectors, membership services, and other coordination services such as locks and barriers. Do not use it for implementing a total ordering service used to order client transactions Recall that etcd is meant to used as a coordination service, and is **not to be used as a database** for storing client related data. See Appendix B.

Containers Use Docker containers to demonstrate your system in a large scale. Docker-Compose might prove itself useful for setting up multiple machines with various roles. You may initially want to test your system via multiple processes on your system before utilizing docker containers.

Data Format There is no specific format for input and output. You are free to choose any suitable format. Assume the syntax is correct; fancy parsers and checks for input errors are unnecessary. Also, there is no need for fancy UI.

Data Storage You may store the data in-memory. No need to save results in persistent memory (disk) or save snapshots in files for after the system was shut down, i.e., no need for DB services.

4.1 Assumptions

Synchrony The system should be asynchronous, but you are allowed to use any type of failure detector (as long you implement it correctly).

You may assume that each client knows the address of all servers in advance, so there is no need to maintain a naming mechanism.

Failure Types In a system with n nodes, there may be up to $f < \frac{n}{2}$ failures nodes. Assume that $n > 3$.

- Assume that there are only fail-stop crashes (with or without recoveries).
  - A crashed node stops sending messages and does not recover during the current execution.
  - No Byzantine faults: nodes cannot behave maliciously, and there are no Sybil attacks from neither the servers nor the clients.
- Assume that the failure pattern is benign, meaning that any node may crash.

5 Writing a Report

Create a detailed external document that describes how you solved the exercise. Your report should include the general architecture of your system and added features, especially ones that you are in-particularly proud of. Also, explain and argue your design choices, and give an approximate location where one can find the implementation of the essential features in your implementation.

Also, if you did not implement some of the required specifications, please note this in your report (be honest!). Also, if you have any major bugs in your code, note them as well. If you have a theory that can explain the bug, please provide it.

6 Grading

The system performance and design, as well as creativity and innovation, will play a major role in the grading consideration.

Also, the report and overall impression of the project is worth 10 points. The use of Dockers is worth 5 points of your grade. The use gRPC of is worth 5 points. Creating a RESTful API of is worth 5 points.

7 Administration & Project Submission

The project must be submitted in singes or pairs. Also please update here that you have found a partner, by adding a "+" next to your ID number. If you are unable to find a partner, please contact the course staff.

Due Date: 04/02/2025 (Last Day of the Semester)

7.1 Submission Structure

Submit you project using the course website. Also, you **must** upload your code to a public GitHub/GitLab[2] repository and provide a link for it in your submission. The link should be in the `repo.url` file with the following structure:

```
[InternetShortcut]
URL=https://example.com/my/repo
```

Your submission must be in a single ZIP file with the following structure:

```
proj.zip
 ├─ report.pdf ............................................ Your report
 ├─ src/ ............................................. Your source code
 │   ├─ docker/.......................................Docker related files
 │   ├─ server/ ........................... Your source code of the server
 │   ├─ etc/ ......................... Any files related for demonstration
 │   └─ README.md ...................... Any comments about the source code
 ├─ students.json
 └─ repo.url
```

[2] Although highly encouraged, you do NOT have to use git for development.

The `students.json` file should have the following structure:

```
[
    {
        "name": "Student Name",
        "id":   123456789
    },
    {
        "name": "Student Name",
        "id":   123456789
    }
]
```

7.2 Live Demo

Each team will present their project to the course staff in a live demo. The available dates and time-slots for presentations will (in the future) be available for selection here. The demos will be held *in* the few weeks after the due date.

A Sample API

A simple key-value store that allows for reading their values and writing updated values to them.

Logical Operation	**Input**	**Consistency**
Getting the list of a keys	Key	Sequentially Consistent or Linearizable
Getting the value of a key	Key	Sequentially Consistent or Linearizable
Getting the value of a multiple or a range of keys	Key List or a Key range	Sequentially Consistent or Linearizable
Updating a key	Key, Value	Linearizable
Updating a list of keys	Key List, Value List	Linearizable
A Read-Modify-Write operation on a key	Key, Operation	Linearizable

B Resources for Using etcd

- Implementing membership with automatic failure detection:
  `https://dev.to/frosnerd/managing-cluster-membership-with-etcd-l0k`
- Watchers:
  `https://github.com/etcd-io/etcd/blob/main/client/v3/experimental/recipes/watch.go`
- Double Barriers:
  `https://github.com/etcd-io/etcd/blob/main/client/v3/experimental/recipes/double_barrier.go`
- Priority Queue:
  `https://github.com/etcd-io/etcd/blob/main/client/v3/experimental/recipes/priority_queue.go`
- Read Write Mutex:
  `https://github.com/etcd-io/etcd/blob/main/client/v3/experimental/recipes/rwmutex.go`