Hash chain replication with RAFT Consensus

Mert Akpinar
mert.akpinar@studio.unibo.it
Yago Illing Arizti
yago.illingarizti@studio.unibo.it

April 4, 2024

This project aims to design and implement a robust application that employs servers to oversee a hash chain across a network of replicated state machines, utilizing the RAFT consensus algorithm [2]. The project includes the development of a user-friendly client capable of appending blocks to the replicated hash chain through a web application interface. The stored data on the hash chain will be in the form of simple text. The primary objectives are to conduct a comprehensive exploration of the RAFT consensus algorithm, study the hash chain data structure, and acquire technical expertise in network application programming. Through this undertaking, the project seeks to contribute to the understanding of distributed systems and consensus algorithms while providing a practical application for managing data in a decentralized and reliable manner.

1 Goal/Requirements

The goal of the project is development of an application consisting of servers which manage a hash chain on a set of replicated state machines using the RAFT consensus algorithm [2]. Development of a client that can append a block to the replicated hash chain over a web application interface is another aim of the project. On the hash chain, simple text data will be stored. The learning outcomes of the project are an in depth study of the RAFT consensus algorithm, the hash chain data structure and gaining technical knowledge about network application programming.

1.1 Scenarios

The user of the resulting software of our project can interact over our web application interface with a cluster of nodes that manage the underlying hash chain data structure, to strengthen their understanding of the raft consensus algorithm. On the mentioned

dashboard, the user can play out various scenarios by starting/stopping individual nodes and appending data to the logs of the nodes, which is then applied to the hash chain as soon as it is replicated on the majority of nodes.

2 Design

2.1 Structure

Our implementation is composed of three main components: the web application dashboard, a gRPC proxy and the raft instance containing the cluster of raft nodes (see figure 1).

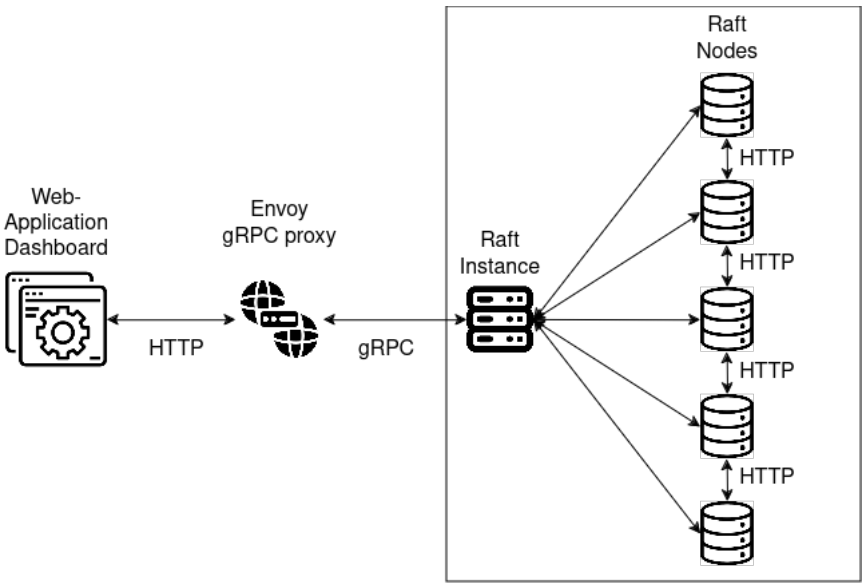


Figure 1: Architecture

Each of the raft nodes manages a hash chain. Each block of the hash chain consists of the hash of the previous block and the entry data (string). The hash of the block is calculated by passing these two values and the index of the block in the hash chain to the SHA256 hashing function. In figure 2 you see a screenshot of the visual representation of the hash chain from our web application dashboard.

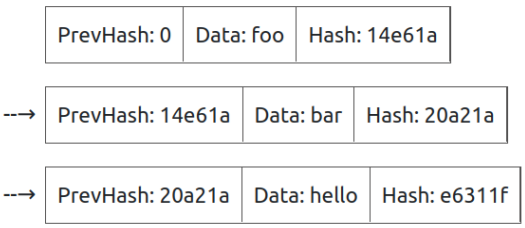


Figure 2: Hash chain data structure visualization

Via the raft consensus algorithm, this hash chain is replicated on each node with a guaranty for data consistency.

2.2 Behavior

The raft instance component initializes a number of raft nodes, which communicate with each other to create consensus. Consensus is created by the nodes by starting an election process to determine a leader (by a majority vote). An election is started when the connection to the leader times out, which is the case at initialization, since there is no leader. This leader is the only node that can receive data from outside the cluster. When the leader receives data (as a string) it first appends to its log (list of received data), broadcasts the newly received data to the follower nodes and then applies the received data to the underlying hash chain, once the log entry is replicated to the majority of the nodes in the cluster. The raft algorithm is implemented on the raft nodes; the raft instance component only serves to provide information of the current state of the nodes to the web application dashboard and to forward a new data entry from the dashboard to the current leader of the cluster.

I will now explain the implemented raft algorithm mechanisms in detail. But before this I will introduce some terminology:

- **term** time frame of arbitrary length which starts with a new election.
- **log** list of data entries received from client/leader which are going to be applied to the hash chain
- **commit index** index of the log, which is replicated on the majority of nodes and can be applied to the hash chain

States of the nodes:

- **leader**: node can receive entries from client and sends them out periodically to other nodes
- **candidate**: node is requesting to be the leader for the current term
- **follower**: node receives entries from leader nodes and can vote for at most one candidate for each term

2.2.1 Voting

The leader of a cluster sends periodically heartbeats to its followers. If a follower does not receive a heartbeat for five seconds plus a random amount of time (up to two seconds), then the node becomes a candidate and increases its term. The candidate then requests a vote from all other nodes. When a follower receives a vote request, then it grants the candidate's vote request if the following conditions are met:

- node is in state follower
- term of candidate matches with term of follower (follower's term is updated to candidate's term if the candidate's term is higher)
- follower has not voted yet
- the term and index of the last log entry sent by the candidate match with the followers log

The sequence of this process is illustrated in figure 3 below, in a cluster containing four nodes:

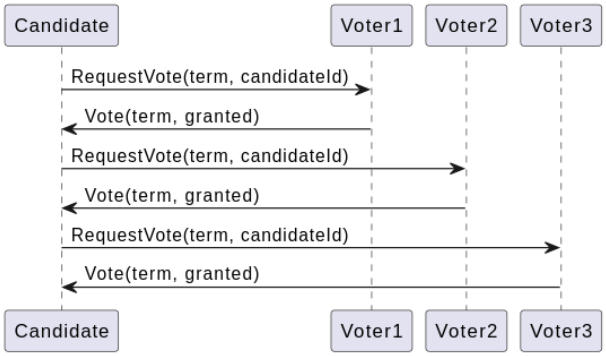


Figure 3: Voting sequence diagram

A candidate becomes the leader, if it receives a granted vote from the majority of nodes. Since a majority of nodes is needed, there can only be one leader for each term. In the case of a split vote, where no candidate wins the election (happens when there are multiple candidates in the same term), the candidates start a new election for a new term after a random timeout. The random timeout decreases the possibility of multiple candidates for the same term. If a candidate receives a request to vote from another candidate for a higher term than its own, it steps down as a candidate and becomes a follower again. The same applies to leaders which discover a term higher term than its own. These state changes are described in the state diagram below in figure 9 from the raft paper [2].

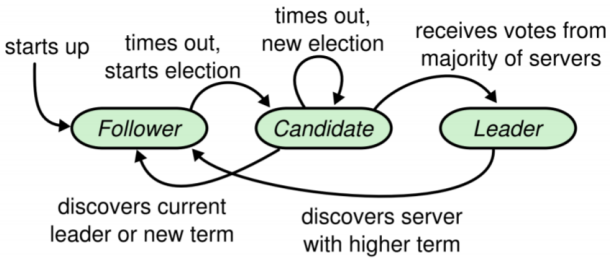


Figure 4: State diagram from [2]

The sequence diagram, below 5, shows an example for the above described scenario of a split vote in a cluster consisting of four nodes where two followers become candidates at the same time (Node1 and Node2). When the split vote occurs, the candidate nodes start a new election after a random amount of time. Node1 first starts the new election and when Node2 receives the request to vote from Node1, it becomes a follower again and grants the vote, since Nodes1's term is higher.

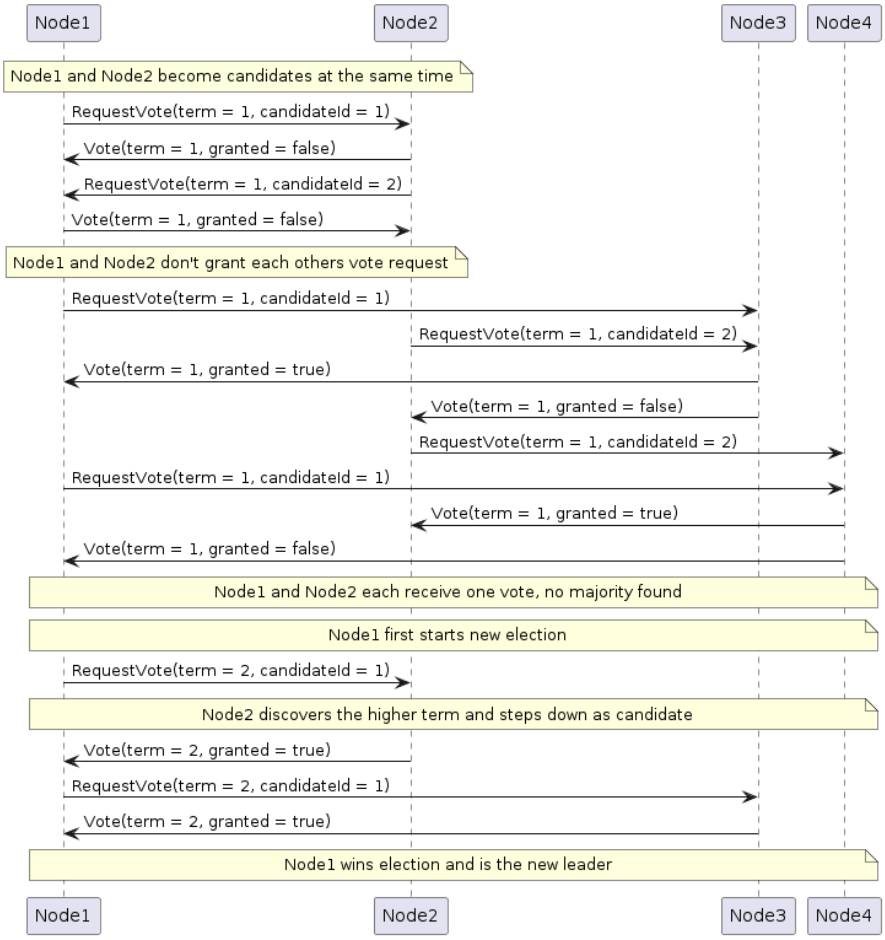


Figure 5: Split vote example

2.2.2 Log replication

Each node has its own log and every entry of the log consists of the index of the log, the data (a string) and the term in which this entry was added to the log. The leader receives from client entries which it appends to this log. The new entries are then periodically broadcasted to the follower nodes by the leader node, so that they can append them to their own log. This is done via the appendEntries RPC, in figure 6 below you can see its parameters.

appendEntries	
term: Int	Leader's term
leaderId: Int	ID of the leader
prevLogIndex: Int	Index of the log entry preceding new entries
prevLogTerm: Int	Term of the log entry preceding new entries
entries[]: List<LogEntry>	Log entries to append
leaderCommit: Int	Index of leader's last committed log entry

Figure 6: AppendEntries RPC

If a follower node receives a appendEntries request containing an entry, it accepts it and appends it to its log if the following conditions are met:

- term of follower matches with term of leader (follower's term is updated to leader's term if the leader's term is higher)
- the term and index of the previous log entry to the one sent by the candidate matches with the followers log last entry

If these conditions are met, then the follower appends the entry to its log at the index after the prevLogIndex (sent by the leader) or overwrites the existing entry and responds to the leader that the operation was a success. An overwrite operation takes place when the conditions are met and the term of the new entry and the existing entry differ. In the case of a successful response, the leader, which keeps track of which index of its log it is going to send next for each follower, increases the next index for the follower that replied with a success response. In the other case, where the conditions for appending the entry to its log are not met, the follower responds to the leader that the operation was unsuccessful. In that case, the leader decreases the index of the entry that is going to be sent next to that particular follower. In other words: the leader goes back in its log until it has reached an entry that can be appended, and then it goes forward in its log and appends the remaining entries. The sequence of requests of this scenario is illustrated in figure7

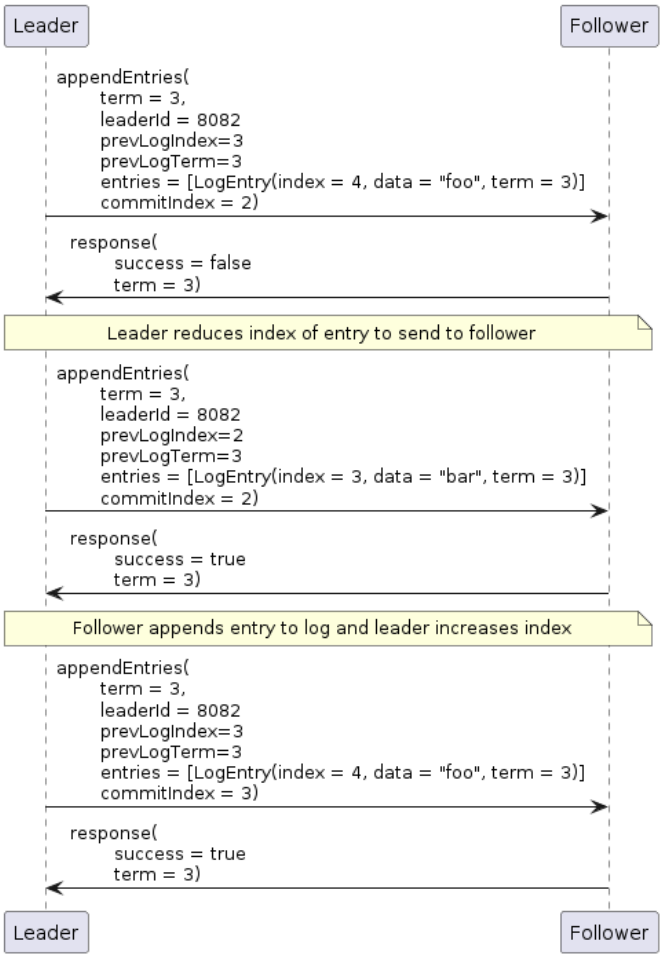


Figure 7: AppendEntry example sequence

2.2.3 Commit index and hash chain

As mentioned before, the commit index indicates the index of the log which is confirmed to be replicated on the majority of node in the cluster. It also indicates until which index it is safe to apply the log entries to the hash chain. If a follower node receives a new entry over the appendEntries RPC and it is able to append this entry (or overwrite an existing entry), then it sets its own commit index to the minimum value of the index of the last applied entry and the commit index sent by the leader. This mechanism serves the purpose that entries sent by the leader are not committed to the hash chain if the leader has not confirmed that they have been replicated on the majority of node in the cluster (in this case, the leader commit index is lower than the last entry index). In the other case (last entry index is lower than leader commit index), this mechanism avoids that entries in the follower's log, that might be overwritten by succeeding entries sent by the leader, are not applied to the hash chain. If the leader sends an empty appendEntries

request (heartbeat) and the prevLogIndex and prevLogTerm match, then the commit index of the follower can be set to the commit index of the leader, since this indicates that the last log entry of the leader matches with the last log entry of the follower.

2.3 Interaction

Since the implementation is composed of three main components; the web application dashboard, a gRPC proxy, and the raft instance that contains the cluster of raft nodes, we need to visualize the interaction between those components in addition to the interaction between nodes.

2.3.1 Web Application Dashboard Interaction with gRPC Proxy

Every gRPC request made with the browser is forwarded to the backend server by the Envoy proxy.

Methods defined in the proto file; Retrieving node information, stopping a node, starting a node, appending an entry to the leader, retrieving hash chain information, go through the proxy, and methods are executed on the server-side. An example sequence diagram can be seen below.

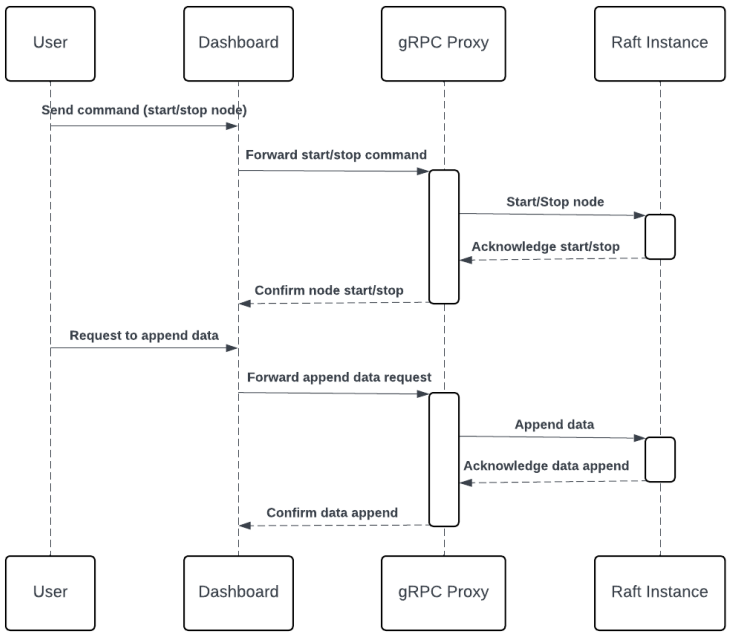


Figure 8: Client - Proxy Sequence Diagram

2.3.2 Raft Instance Interaction with Nodes

As stated in the previous section, every method call made from the browser has an action handler defined in Raft Instance. The Raft Instance can access the current status of the nodes, and manage them according to the incoming request.

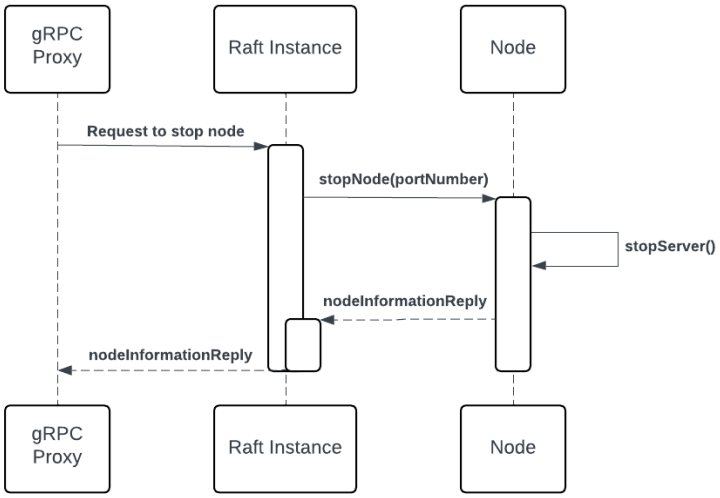


Figure 9: Interaction between the Raft Node and Raft Instance - Stop node command

3 Implementation Details

3.1 Web Application Interface

The web application interface for our system is built using React.js, a popular JavaScript library for building user interfaces. One significant aspect of our implementation is the seamless integration between the frontend and backend components. To achieve this, we utilized a Protocol Buffer (proto) file, which is generated in the backend application, to automatically generate JavaScript files for the frontend project. This approach enabled us to directly call the methods defined on the server from our React application.

In addition to utilizing the generated JavaScript files for communication with the backend, we employed the Envoy proxy to facilitate gRPC communication between our React app and the Kotlin server application. The decision to use Envoy proxy was driven by its support for both HTTP/1.1 and HTTP/2 protocols. It also supports gRPC, which is based on the HTTP/2 protocol. It is used as the routing and load-balancing substrate for gRPC requests and responses.

3.2 Server Implementation

The RaftInstance class is the entry point of the server application. The gRPC bassed communication is supplied by an inner class called "RaftInstanceService". This class extends abstract "CommunicationServiceCoroutineImplBase" class that has been auto-generated by the compilation of the proto file. Method bodies are defined in that class and are used for client - server communication. That inner class also extends "BindableService" interface to be added to RaftInstance as a service and to enable gRPC communication. Besides that, each node is added to the RaftInstance as a service and operates on its own ports.

3.2.1 Raft Node Implementation

For the raft nodes, we have used Ktor, which is a framework for building asynchronous servers with the Kotlin programming language [1]. By defining routes for requesting a vote and appending an entry, nodes are able to communicate with each other over HTTP requests. Each HTTP request is processed sequentially by the node. Periodical requests to other nodes, like the heartbeat from the leader node to the follower nodes, are implemented with Kotlin Coroutine Jobs. We also used these Jobs to implement timers that trigger an action after a certain amount of time if not canceled, for example, when a follower does not receive a heartbeat within 5 seconds it will start becoming a candidate.

4 Self-assessment / Validation

We developed six integration tests that validate the behaviors of the implemented raft consensus algorithm described in section 3.2.1 to 3.2.3. The tests validate the following properties / mechanisms and check that node properties like the log, hash chain, term and commit index are correct:

- nodes elect one leader
- if a leader goes offline, a new leader is elected
- inconsistent/uncommitted log of old leader node is overwritten by new leader
- leaders correctly broadcast new entries
- commit index is only updated if the log entry is replicated on the majority of nodes in the cluster
- log of a restarted node catches up to the current state of the log

These tests can be run by running the following commands in the terminal after cloning the project repository.

Listing 1: Run Tests

```
cd ktor-raft
./gradlew :test
```

These tests and an additional job that checks if the build of the gradle project succeeds are embedded in a Gitlab continuous integration pipeline that we created. This pipeline is executed on a push to the Gitlab instance. When developing our software, we developed new features in separate git branches and merged these with the master branch over the Gitlab interface, which showed the status of the CI pipeline. By proceeding in this manner, we were able to have certainty that the requirements were still being fulfilled with each incremental change to the software. Unfortunately, we did not create tests that validated the communication between the web application dashboard with the raft instance via the gRPC proxy. But extensive usage and testing of the web application, makes us confident that it works as expected.

5 Deployment Instructions

After cloning the project repository follow these steps to run the three components.

The server application containing the cluster of raft nodes is embedded in a Gradle wrapper, and it is therefore not necessary to have Gradle installed. To run the raft instance (default configuration with 5 nodes in the cluster), run the following in your terminal:

Listing 2: Raft instance

```
cd ktor-raft
./gradlew :run
```

You can change the number of nodes in the cluster (e.g., to 7 nodes) by running:

Listing 3: Custom number of nodes

```
./gradlew :run --args="-numberOfNodes 7"
```

To build and run the web-application on your machine you need to have docker, docker-compose (for windows) and NPM (node package manager) installed on your machine.

To build and run the docker container containing the gRPC proxy, you need to run the following commands on a Linux based system:

Listing 4: gRPC proxy on Linux

```
cd web-app/
docker build -t my-grpc-container:1.0.0 src/linux/
docker run --net=host my-grpc-container:1.0.0
```

and on a system running windows:

Listing 5: gRPC proxy on Windows

```
cd web-app/src
docker-compose up
```

Finally, to run the web application containing the monitoring dashboard run these commands in your terminal:

Listing 6: Web application dashboard

```
cd web-app/
npm i
npm start
```

And you're done! The monitoring dashboard should be running on http://localhost:3000 showing a widget for each node of the raft cluster.

6 Usage Examples

This section shows the usage of the web application dashboard and its functionalities. When the raft instance is running, you can see for each node a widget like the one seen below in figure 10. The widget shows for each node the current state, term and commit index.

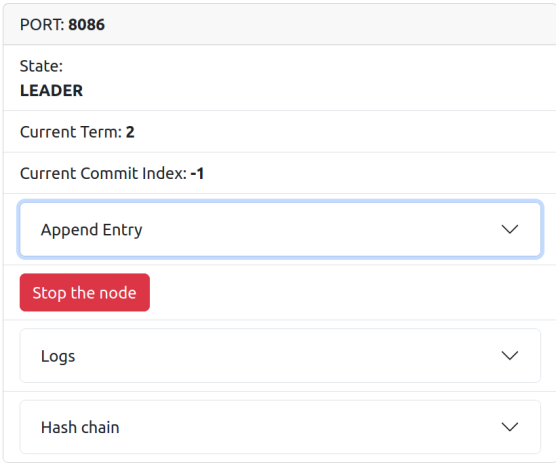


Figure 10: Node widget

The widget for a leader node additionally shows the append entry field that opens up once it is clicked (see figure 11). After filling out the field the user can press the "send" button, to append the entry to the log of the leader node, which will try to broadcast the new entry to the follower nodes.

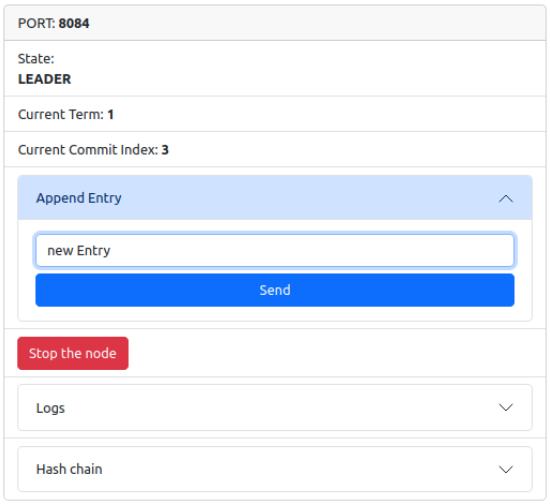
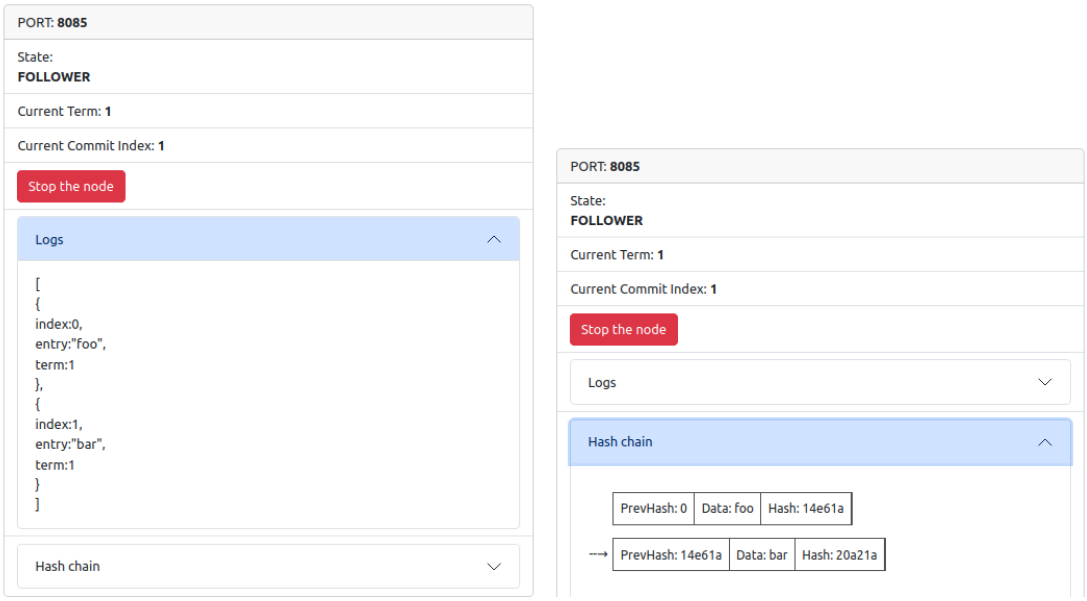


Figure 11: Append entry field for leader node

To view the current log or hash chain of a node, the user can click on the according field as shown in figure 12.



(a) View node log

(b) View node hash chain

Figure 12: View log and hash chain

By pressing the red button with the "Stop the Node" label, it is possible to disable the node temporarily, to simulate outage scenarios. If the node is stopped, then its state is changed to "Inactive" and the button turns green and changes the label to "Start the node" (see figure 13). If this button is pressed, then the according node is restarted.

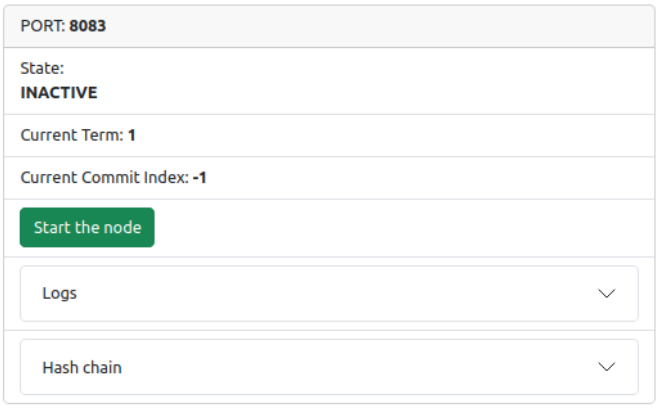


Figure 13: Stopped node

7 Conclusions

In conclusion, this project has successfully achieved its objectives of designing and implementing a robust demonstration application utilizing servers to manage a hash chain across different instances of replicated state machines, employing the RAFT consensus algorithm. Additionally, a user-friendly client has been developed to facilitate block appending to the replicated hash chain via a web application interface, with stored data in the form of simple text. All the features specified in the Raft paper have been implemented, and situations such as communication between nodes, data integrity, and how to proceed in case of node crashes have been tested.

7.1 Future Works

In its current state, the application is a demonstration of the raft consensus algorithm in a distributed system, but all the nodes of the cluster run on the same machine. To make this software a real distributed system, it's necessary to modify the software, so each node can be run on a different machine on the network. First, you would modify the software to only launch one node and passing a list of IP addresses and ports of each node in the network, on startup. This list must be complete, and there would be a need for a mechanism to ensure that all nodes are aware of the same nodes in their network. It is also possible to add a discovery mechanism where nodes are dynamically added and removed from the network. The web application would then be needed to be adapted accordingly. The hash chain data structure could be expanded to store transaction data in each block of the hash chain and the possibility to append transactions from a client to create a prototype of a cryptocurrency.

7.2 What did we learn

In this project, we studied in depth the raft consensus algorithm and the complex problems that are related to creating consensus in a network. Additionally, we learned about

how to implement gRPC stubs by defining them in proto files.

References

[1] Ktor framework. `https://ktor.io/`. Accessed: 2024-04-03.

[2] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, page 305–320, USA, 2014. USENIX Association.