Note

I plan to upload this assignment over a github once grades are out. I wrote the readme considering that fact. Reading it in plain text editor might not be a good idea! I suggest to read readme.pdf instead

Project Title

Fail-Silent Replicated Token Manager with Atomic Semantics

Project Description

Implemented a distributed system for managing tokens. Existence and Replication schemes of all tokens are static and known apriori to all the server and client nodes in your system. All tokens are initially created and replicated by the token management launcher which also initializes all the serves. Further, Clients can issue RPC calls to the server to execute read or write methods on tokens.

Token

A token is an abstract data type, with the following properties: id, name, domain, and state. Tokens are uniquely identified by their id, which is a string. The name of a token is another string. The domain of a token consists of three uint64 integers: low, mid, and high. The state of a token consists of two unit64 integers: a partial value and a final value, which is defined at the integer x in the range [low, mid) and [low, high), respectively, that minimizes h(name, x) for a hash function h. Hash function used in SHA-256.

Supported Operations

- **create(id):** create a token with the given id. Return a success or fail response.
- **drop(id):** to destroy/delete the token with the given id
- **write(id, name, low, high, mid):**
  1. set the properties name, low, mid, and high for the token with the given id. Assume uint64 integers low <= mid < high.
  2. compute the partial and final value of the token
  3. Partial value is min H(name, x) for x in [low, mid)
  4. Final value is min(Partial Value, min H(name, x) for x in [mid, high))
  5. Return the final state of the token consisting of partial value and final value on success or fail response otherwise
- **read(id):** returns token state value on success or fail response otherwise

Setup Environment

Go Installation

Follow: Download and install Go

Use Version: go1.17.7

Protocol Buffers Installation

Follow: Protocol Buffer Compiler Installation

Use Version: 3

Install gRPC plugins

```
$ go install google.golang.org/protobuf/cmd/[email protected]26
$ go install google.golang.org/grpc/cmd/[email protected]1
```

Update PATH for protoc

```
$ export PATH="$PATH:$(go env GOPATH)/bin"
```

Run

To lauhch the Token Management System:

```
# go to project directory
cd <project_directory>
e.g. $ cd /Users/aditya/Documents/Courses/AOS/CMSC621_project3

# start token management system
$ go run tokenmanager_launcher.go

# To use different configuration YAML than the default one
go run toknenmanager_launcher.go -yaml <file_name>
e.g. $ go run tokenmanager_launcher.go -yaml configuration.yaml
```

Server (User is not expected to launch servers explicitly at all for this project):

```
# go to project directory
cd <project_directory>
e.g. $ cd /Users/aditya/Documents/Courses/AOS/CMSC621_project3

# start server
go run server.go -host <server_addresss> -port <port_number>
e.g. $ go run server.go -host localhost -port 50051
```

Client (User is not expected to fire create and drop requests explicitly at all for this project):

```
# go to project directory
cd <project_directroy>
e.g. $ cd /Users/aditya/Documents/Courses/AOS/CMSC621_project3

# create request
go run client.go -create -id <id_num> -host <host_addreess> -port <port_number>
e.g. $ go run client.go -create -id 1 -host localhost -port 50051

# write request
go run client.go -write -id <id_num> -name <token_name> -low <low> -mid <mid> -high <high> -host <host_address>
 -port <port_number>
e.g. $ go run client.go -write -id 1 -name abcd -low 1 -mid 5 -high 10 -host localhost -port 50051

# read request
go run client.go -read -id <id_num> -host <host_address> -port <port_number>
e.g. $ go run client.go -read -id 1 -host localhost -port 50051

# drop request
go run client.go -drop -id <id_num> -host <host_address> -port <port_number>
e.g. $ go run client.go -drop -id 1 -host localhost -port 50051
```

To run the demo:

```
# go to project directory
cd <project_directroy>
e.g. $ cd /Users/aditya/Documents/Courses/AOS/CMSC621_project3

# set executable permission for demo script
$ chmod +x ./demo_proj3.sh

# execute demo script
./demo_proj3.sh
```

To check the server and tokenmanager_laucher's logs check output directory from where the bash script was launched

Project Files and Directories

- **server.go:** Code for the server operations (Not much change from project 2)
- **client.go:** Code for the client operations (Not much change from project 2)
- **tokenmanager_launcher.go:** A wrapper code to read initial replication configuration from YAML, and start servers and crate tokens (This file is newly introduced in project 3)
- **configuration.yaml:** YAML file with the replication scheme of all the tokens, i.e. an array
  token: <id>
  writer: <access-point>
  readers: array of <access-point>s
  where <access-point> is of the form <ip-address>:<port>, whereas a writer may also be a reader.
- **token:** Directory containing code related to token management like proto definitions and logic for each operation that can be performed on tokens (Major changes for project 3 in token.go and token.proto)
- **utils:** Directory containing utilities and helper functions (Not much change from project 2)
- **go.mod:** Root dependecny managment
- **go.sum:** Checksum for dependencies
- **demo_proj3.sh:** Shell script demonstrating 4 different scenarios of the project 3
- **demo_proj2.sh:** Can be Ignored (This file is from project 2)
- **analysis_helper.sh:** Lists commands I generally used to analyze my outputs and it is sometimes to difficult to navigate stdout and stderr through large pile of text (Created for project 2, Not sure about relevancy for project 3, may help, Did not find the need of of one for project 3)
- **output:** Directory which stores redirected logs from servers and tokenmanager_launcher ran via demo script. This folder contains 4 subdirectories clasifying output logs for each use case
- **demo_screenshots:** Screenshots of the demo I ran

Code Description - What Did I do/Assumptions Made/Deviations

I believe the code itself is very well commented and readable. Especially, I felt the need of comments to demonstrate how I implemented read-impose-write-all (quorum) so, I added number of comments which gives very clear picture of each step. I highly recomment going through code of token.go

I tried to note few more things below:

- First I moved the final value calculation to write and updating both partial and final values in the write itself. Keeping the implementation of project 2 essentially converts read to write, and doesn't make much sense to me. (I believe this was also discussed in the class, and professor said this expected, Also had a discussion with TA around this)

- One of the major deviation I took from what is asked in the project is to use min values instead of argmin for read and write operations

- Reason being argmin will always result in same partial and final values. (I discussed this already with the TA, and reconfirmed for project 3)

- The project starts with execution of tokenmanager_launcher.go which reads the initial configuration of YAML, launches all unique servers (access points) from that file, and create tokens on reader and writer nodes as mentioned in the configuration.yaml, and just sleeps in a loop. To stop all servers just halt the execution of tokenmanager_launcher.go by pressing Ctrl+c

- For the requirement of writer can be reader, check token with id 1

- Next the major changes pertaining to Project 3 as opposed to project 2 are in the proto definitions and read/write APIs

- Assumption is once the tokens are created at all nodes by tokenmanage_launcher, user will not create new tokens or drop them in between. Create and Drop APIs are mostly carry forwarded as it is from project 2. So, you might be able to create and drop tokens, but you are not expected to (And I didn't test it extensively)

- For read and write, I have implemented read-impose-write-all quorum, the quick summary of write and read is as follows

- **Write:** Accepts the request --> Check if token is available in store (fail otherwise) --> Check if the node is previliged for write (fail otherwise) --> Acquire lock (only on single resource, not entire store, In order to server parallel requests) --> Calculate partial and final values --> Send parallel write broadcast request to readers containing id, domain, state, timestamp, and reading flag (set) --> Check if majority achieved i.e. acks > ((N+1) / 2) --> As soon as majority achieved update token store and respond to client

- **Read:** Accepst the request --> Check if token is available in store (fail otherwise) --> Acquire lock (only on single resource, not entire store, In order to server parallel requests) --> Check if the node is previliged for read (fail otherwise) --> Send parallel broadcast read requests to all readers containing only id --> Check if majority achieved i.e. acks > ((N+1) / 2) --> As soon as majority achieved, find the reader who reported highest timestamp --> Write back i.e. Send parallel write broadcast request to readers containing id, domain, state, timestamp, and reading flag (set) --> Check if majority achieved i.e. acks > ((N+1) / 2) --> As soon as majority achieved update token store and respond to client

- **WriteBroadcast:** It updates the token replicas' timestamp, domain, and state either if the broadcast request's timestamp is latest than replica's copy or if the reading flag is set i.e. in case of write back by read

- **ReadBroadcast:** It just returns the domain, timestamp, and state associated with the token

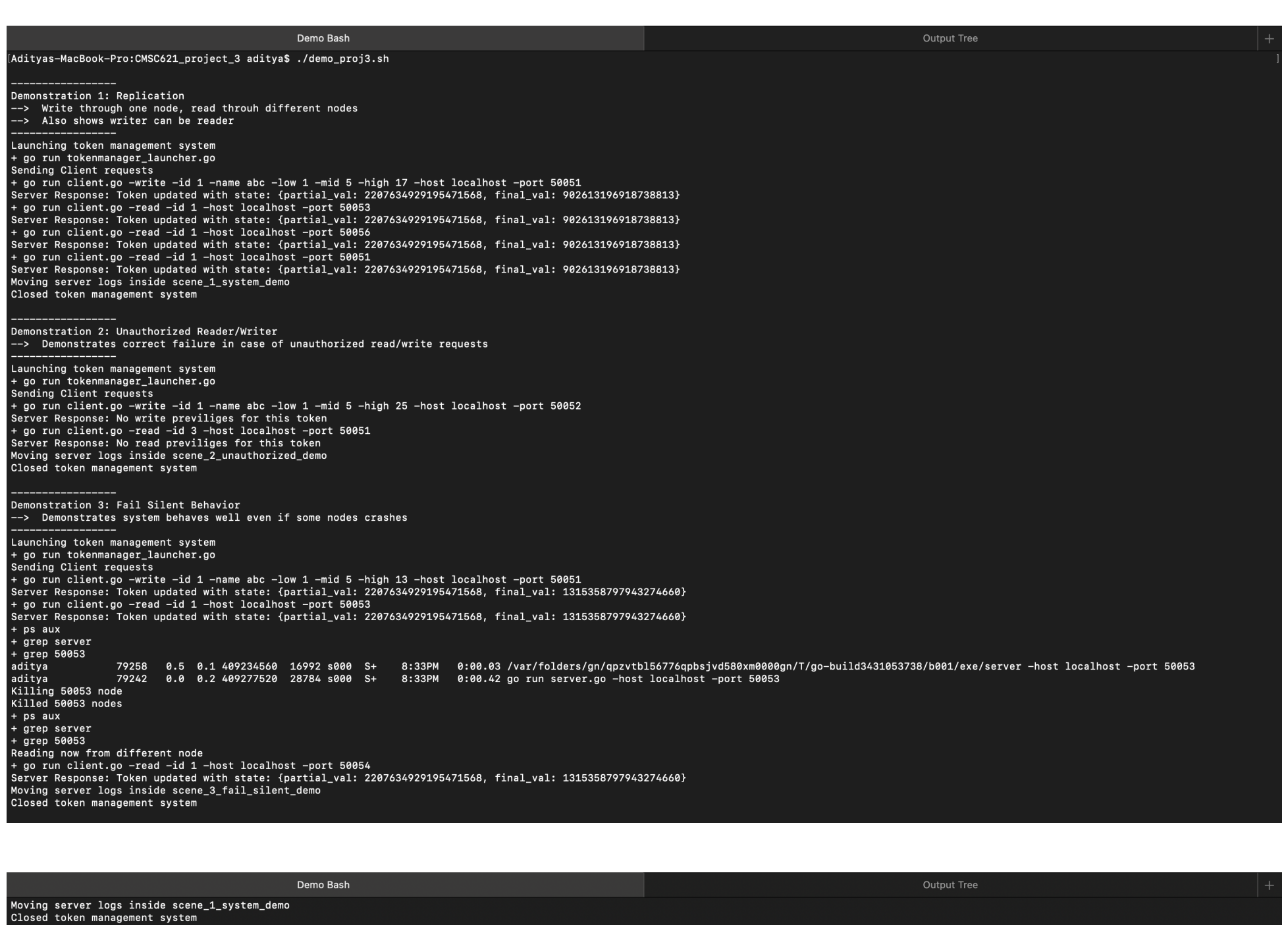
- To achieve the majority for both even and odd cases, I used (N+1) / 2

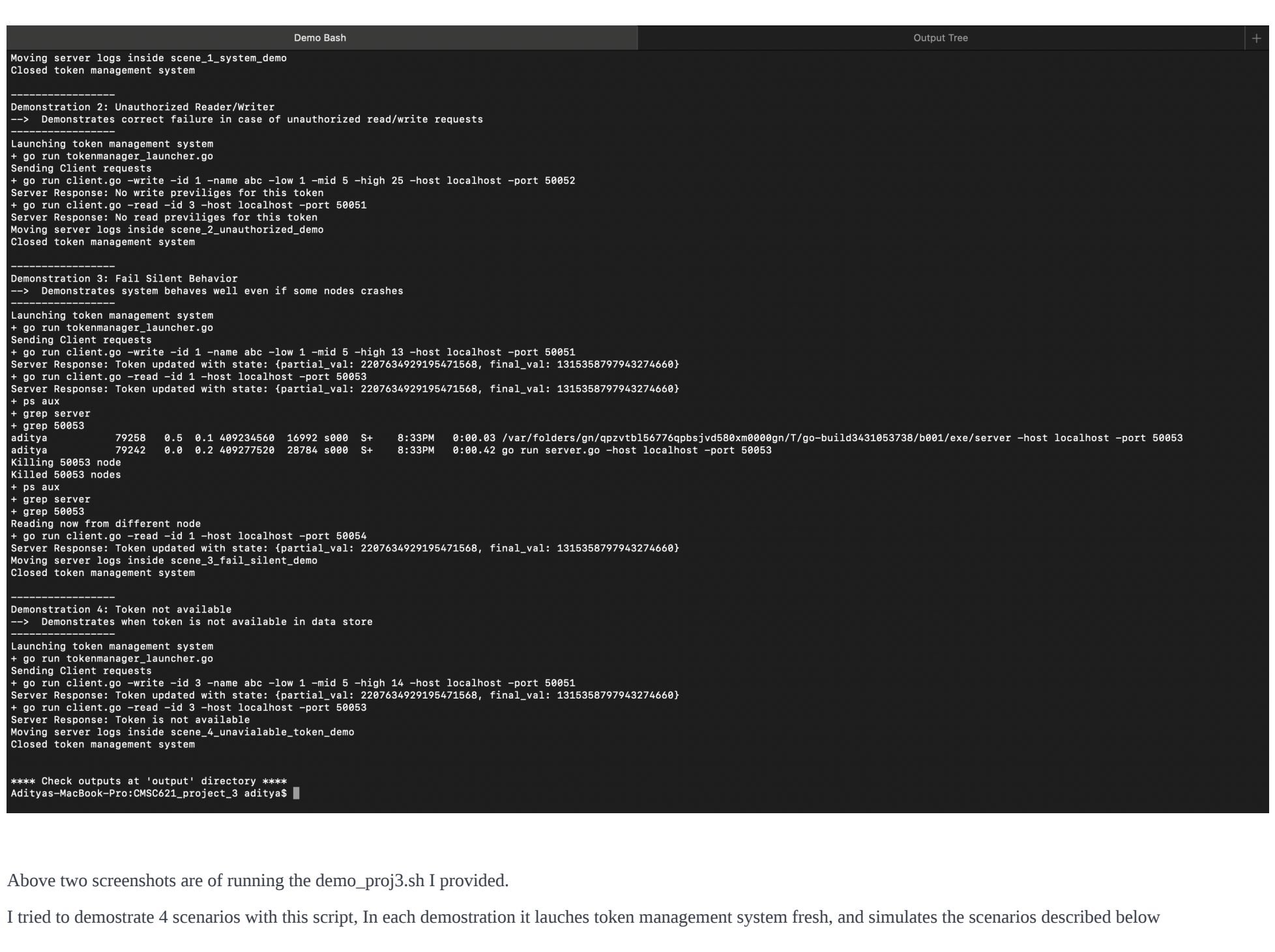
- To raise parallel broadcast requests, I just made rpc calls inside go routine, and updated a common channel to get real time acknowledgements

- While supporting concurrency, my code supports following type of operations
  – Operations with different id - Parallel execution
  – Read operations with same id - Parallel execution
  – Any other combination of requests with same id - Serial execution

I hope this clears! Please let me know if you have any questions about the implementation

Demo



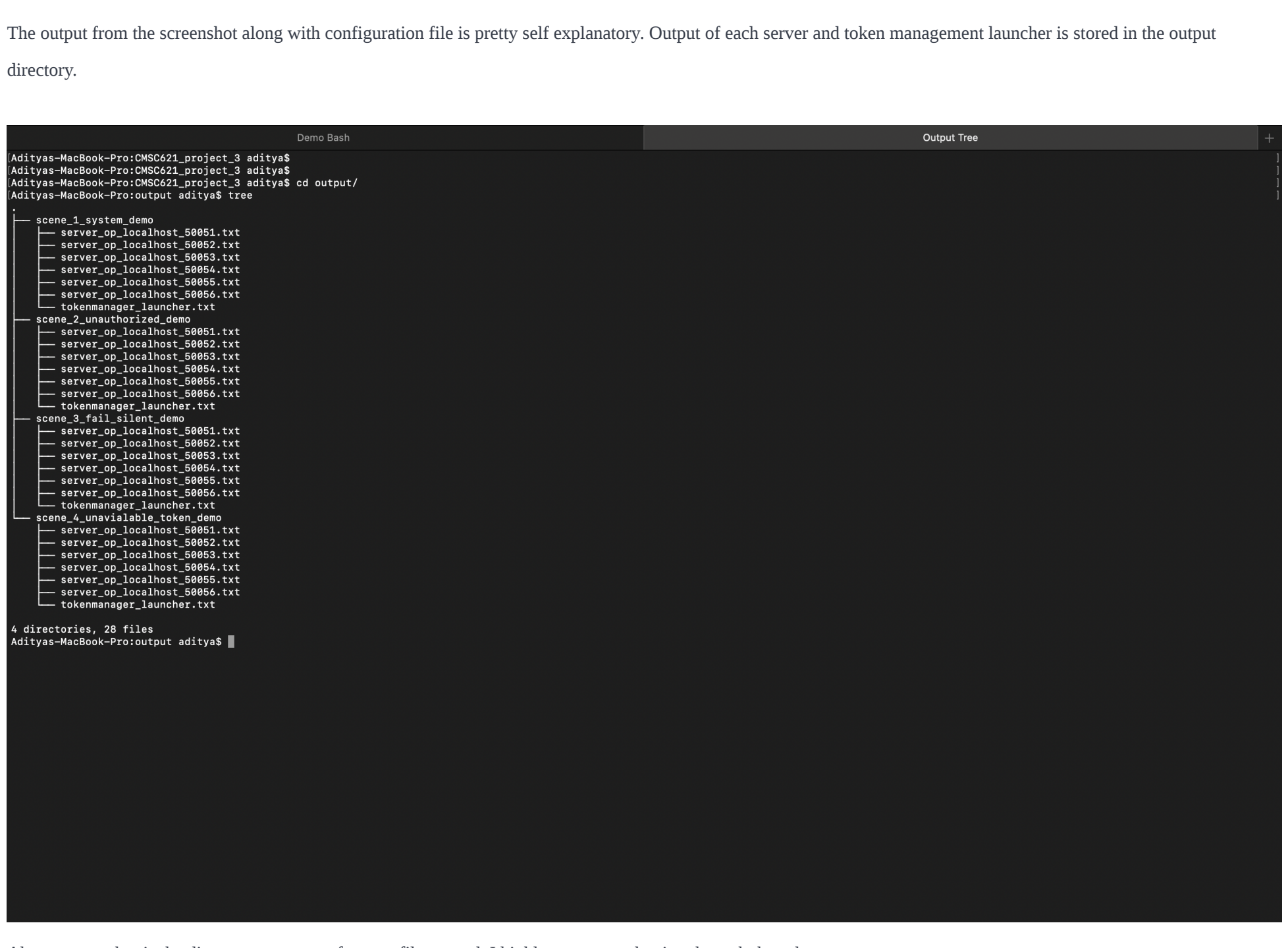


Above two screenshots are of running the demo_proj3.sh I provided.

I tried to demostrate 4 scenarios with this script, In each demostration it lauches token management system fresh, and simulates the scenarios described below

1. **Replication:** Write from one node, read from other nodes
2. **Unauthorized Reader/Writer:** If node is not authorized to read/write, but it has token (because it can be reader but not authorized to write it or vice-a-versa) should not read/write
3. **Fail Silent Behavior:** If one node crashes, system should still function, and read from other nodes should function
4. **Token not available:** If the token is not available at the node beacuse it is neither reader nor writer, It should fail gracefuly

The output from the screenshot along with configuration file is pretty self explanatory. Output of each server and token management launcher is stored in the output directory.



Above screenshot is the directory structure of output files stored. I highly recommend going through these logs

Few things to notice in the server logs (check ports from the screenshots to identify interesting server nodes):

Scenario 1:

- Check the configuration file to know which are reader and writer nodes for the token
- Even though number of reader nodes is more but once majority is achieved servers respond, proves majority voting (quorum)
- Sequence of broadcast request raised is different than responses and there are random response in between read/write requests as well which proves parallelism of broadcast requests
- In the read request there are timestamps collected and write-back queries which proves read-impose-write-back

Scenario 2:

- Map the accesspoints against configuration file which proves they are not authorized for respective write and read requests

Scenario 3:

- Especially observe that I killed the server running on 50053 in between (observe grep output and check bash script)
- That node was one of readers and earlier responded to read query
- It died, still 50054 responded correctly with the value
- Check the file output/scene_3_fail_silent_demo/server_op_localhost_50054.txt, It clearly shows that server running on 50053 is not available

Scenario 4:

- Map the accesspoints against configuration file which proves 50051 is not authorized to read as well as write for token 3, and hence doesn't have it available

References

From Project 2

- https://forum.golangbridge.org/t/cannot-import-package-variables-to-main-package/21193/2
- https://github.com/evilsocket/opensnitch/issues/373#issuecomment-803663343
- https://github.com/grpc/grpc-go/issues/3794#issuecomment-720599532
- https://stackoverflow.com/questions/15178088/create-global-map-variables
- https://tutorialedge.net/golang/go-grpc-beginners-tutorial/
- https://yourbasic.org/golang/errors-explained/
- https://go.dev/blog/maps
- https://www.geeksforgeeks.org/math-inf-function-in-golang-with-examples/
- https://learnandlearn.com/golang-programming/golang-reference/golang-find-the-minimum-value-min-function-examples-explanation
- https://yourbasic.org/golang/multiline-string/

For Project 3

- https://zetcode.com/golang/yaml/
- https://www.sohamkamani.com/golang/sets/
- https://www.geeksforgeeks.org/how-to-split-a-string-in-golang/
- https://stackoverflow.com/questions/37122401/execute-another-go-program-from-within-a-golang-program
- https://stackoverflow.com/questions/28322997/how-to-get-a-list-of-values-into-a-flag-in-golang
- https://medium.com/@tzuni_eh/go-append-prepend-item-into-slice-a4bf167eb7af
- https://golangdocs.com/list-container-in-go

** PS: Apologies for spelling/grammar mistakes, wrote this readme at last minute **