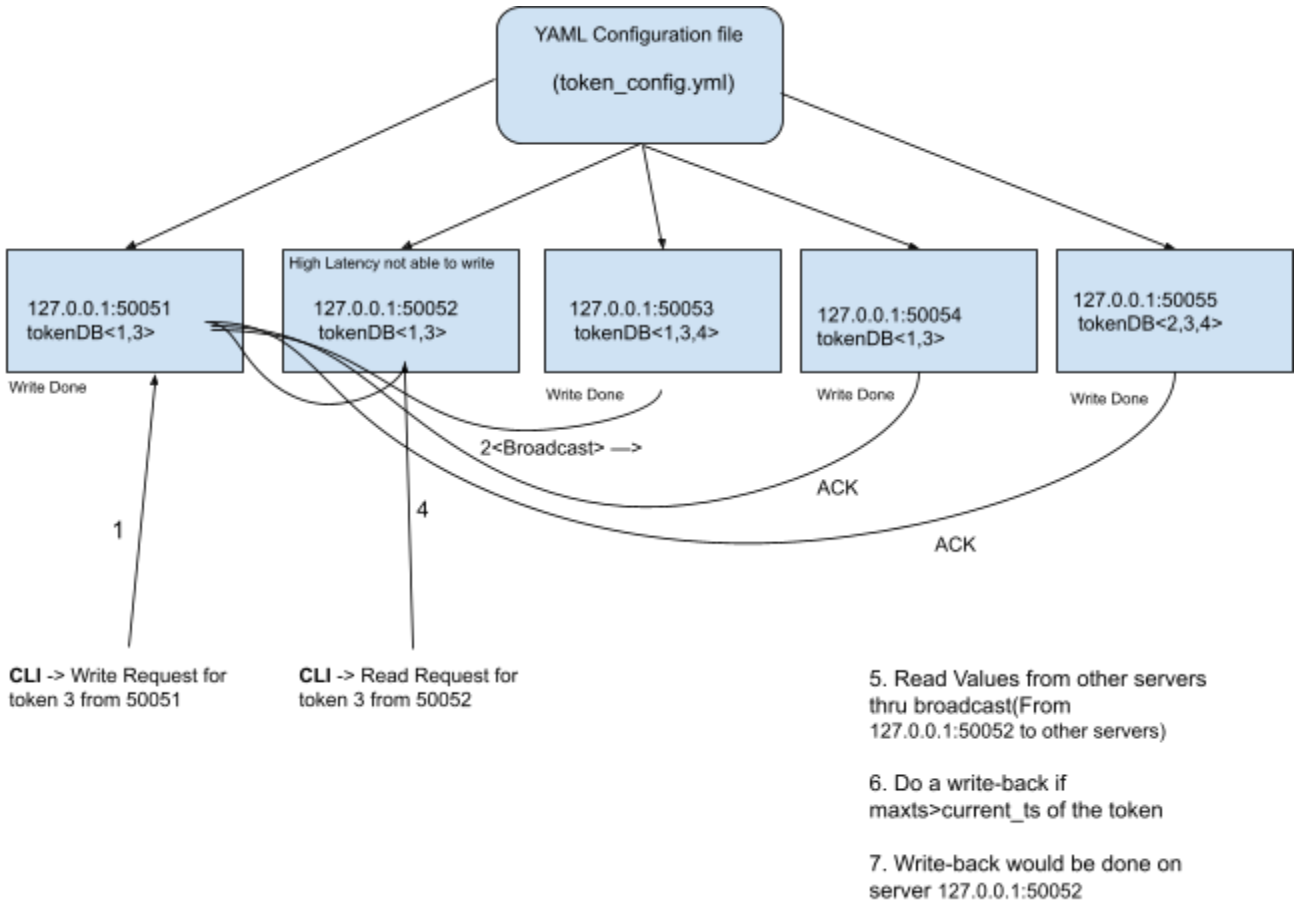
Fail-Silent Replicated Token Manager with Atomic Semantics

1. Introduction

This project is basically the extension of the previous project. Here, we have added functionalities to support replication and atomic semantics. We have simulated a fail-silent model on the tokens which are maintained by the individual servers according to the access given to them. The access points and the token ids which should be maintained are being read from the YAML file. This configuration is then applied to the servers when they are spawned. Additionally, the replication scheme is maintained using the YAML file. Once the configuration has been read and servers are spawned the client would now start requesting write and read requests. The authorization would be checked before serving the requests. For example, if a client gives a write request to the server which doesn't have the token mentioned or is not the writer node then that request would fail to run. In this project, I've used partial_value, name, low, mid, high which can be written and read from the authorized servers.

To maintain the atomicity and replication scheme I've used the read-impose-write-all(Quorum) protocol. In this protocol, the write request would first broadcast the values written in the request to all the reader nodes. Once it gets the acknowledgement from the majority of the servers(like from 2 servers out of 3) it would return the write value. Now, if the request is made to the reader server it would ask for the values stored in other servers. After getting the values it will check for the value with maximum timestamp. Now, we would have the latest value which was written in the servers. Once we get the latest value we'll do the write-back across other nodes. This implementation has been shown in the demo.

2. Architecture



3. Implementation/Documentation

Following functionalities has been added to this project.

```
rpc WriteBroadcast(WriteBroadcastRequest) returns (WriteBroadcastResponse) {}
 rpc ReadBroadcast(ReadBroadcastRequest) returns (ReadBroadcastResponse) {}
```

```
message WriteBroadcastRequest {
 uint64 hash_val=1;
 bool reading_flag=2;
 int32 ack=3;
 string wts=4;
 string name=5;
 uint64 low=6;
 uint64 mid=7;
 uint64 high=8;
 int32 token_id=9;
 string server=10;
}
message WriteBroadcastResponse {
 int32 ack=1;
}
```

```
message ReadBroadcastRequest {
 int32 token_id=1;
 // string server=6;
}
message ReadBroadcastResponse {
 string wts=1;
 uint64 finalVal=2;
 string name=3;
 uint64 low=4;
 uint64 mid=5;
 uint64 high=6;
}
```

```
func (s *TokenManagerServer) WriteToken(ctx context.Context, in *pb.WriteTokenMsg)
(*pb.WriteResponse, error) {
```

This function WriteToken first writes the values for the token that are passed by the client. After writing these values, we need to broadcast these updated values along with the latest timestamp to the other nodes in the system. This function does exactly that.

```
func startBroadcast(partial_val uint64, read_name string, read_high uint64, read_mid
uint64, read_low uint64, len_readlist int, wts string, server string, reading_flag
bool, cnt *int, cnt_ch chan int, token int32)
```

This startBroadcast function is basically a goroutine used for broadcasting asynchronous rpc calls. These rpc calls are basically write calls to other nodes in the system. The channel returns the number of acks received back to the WriteToken function which keeps a count of the number of acks received.

```
func (s *TokenManagerServer) WriteBroadcast(ctx context.Context, in
*pb.WriteBroadcastRequest) (*pb.WriteBroadcastResponse, error)
```

The WriteBroadcast function is the function that is called by the asynchronous write broadcast call. This function writes the latest values of the token and returns an ack.

```
func (s *TokenManagerServer) ReadToken(ctx context.Context, in *pb.Token)
(*pb.WriteResponse, error)
```

The ReadToken function issues a broadcast for all the reader nodes based on the read-impose-write-all protocol. This broadcast receives the values of the tokens along with the timestamp as an ack.

```
func startReadBroadcast(token_id int32, wts string, server string, readlist_ch chan
readlist) {
```

This is the actual goroutine that is called while broadcasting a read request. The acks received are passed through a channel to the ReadToken function for further evaluation.

```
func (s *TokenManagerServer) ReadBroadcast(ctx context.Context, in
*pb.ReadBroadcastRequest) (*pb.ReadBroadcastResponse, error)
```

The ReadBroadcast is the delivery function for the startReadBroadcast function. It returns the value of the appropriate token as an acknowledgement.

4. Demo

- Running the code

  Please run the following command if needed -

  go mod init

  To compile the proto buffer file. Run the following command -

  protoc --go_out=. --go_opt=paths=source_relative --go-grpc_out=. --go-grpc_opt=paths=source_relative token_management/token_pb.proto

  token_config.yml *→ This file contains the configuration which would be read by servers.*

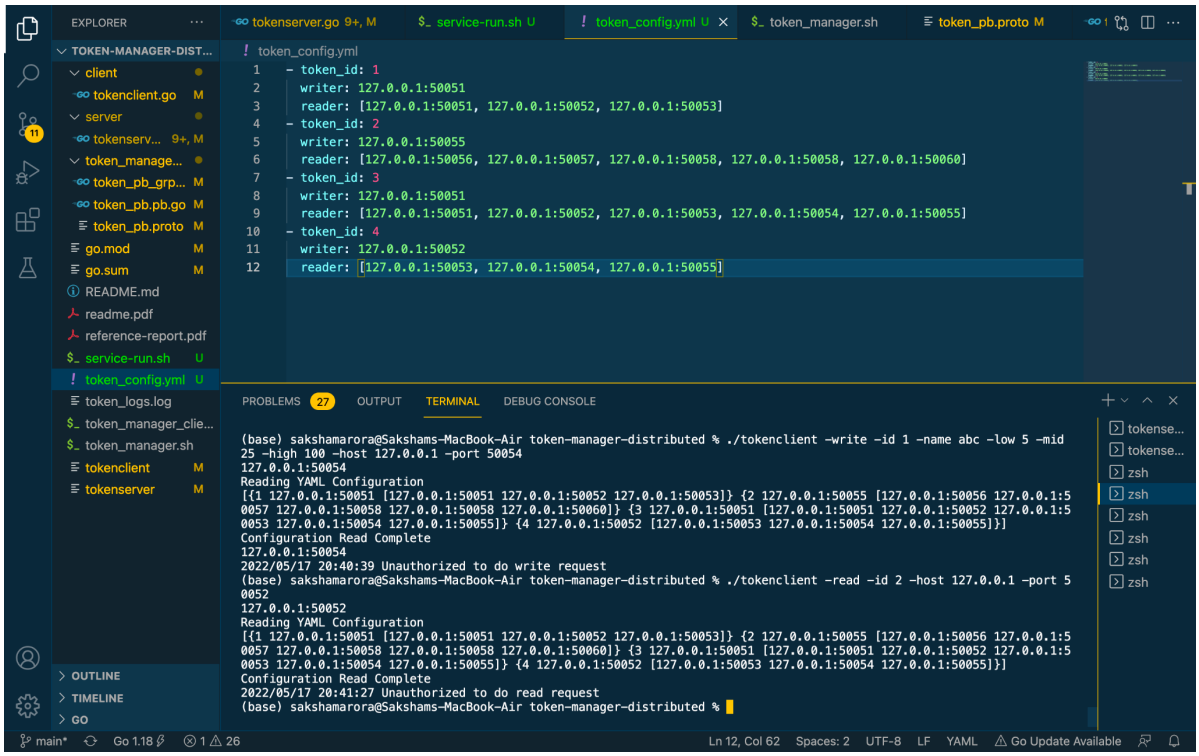
  To run servers individually. Here by default the IP address is 127.0.0.1(i.e localhost). Please change port number as needed.

  For Example -
  go build server/tokenserver.go → Terminal 1
  ./tokenserver -port 50051 → Terminal 1
  ./tokenserver -port 50052 → Terminal 2
  ./tokenserver -port 50053 → Terminal 3

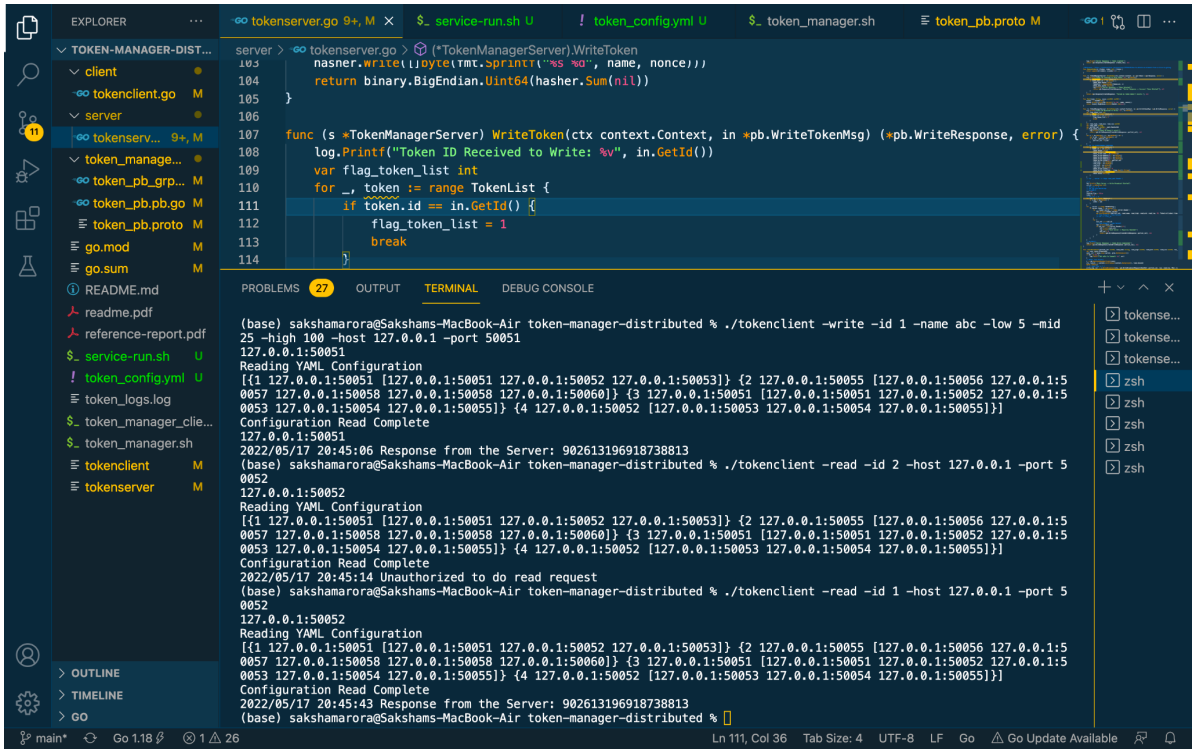
  Terminal 4 →
  go build client/tokenclient.go
  ./tokenclient -write -id 1 -name abc -low 5 -mid 25 -high 100 -host 127.0.0.1 -port 50051
  ./tokenclient -read -id 3 -host 127.0.0.1 -port 50052

  ./service-run.sh *→ This bash script will run the above configuration and will kill the servers once the request is served.*

- Demonstrating scenario where Read-Write authorizations are checked



Below all the necessary servers have already been listening.



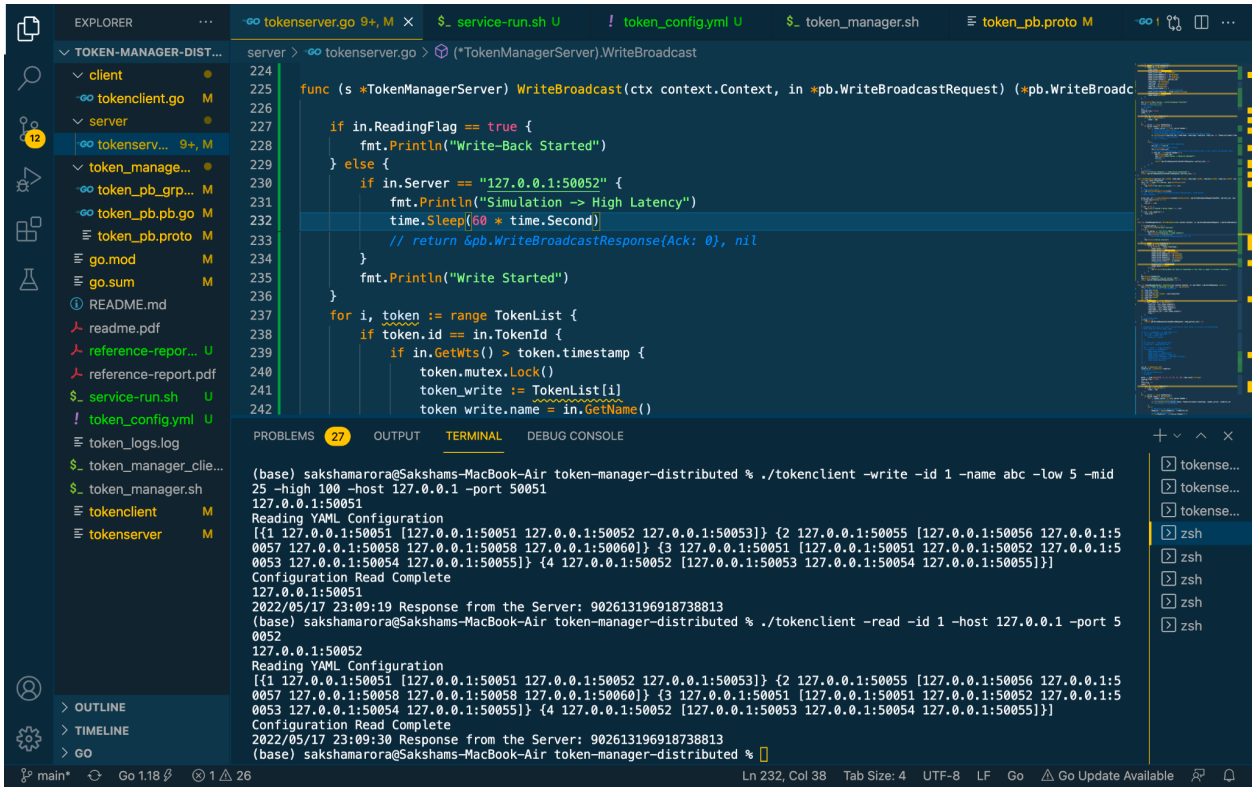
- Demonstrating the fail-silent model.

  Here I'll be taking the scenario where out of 3 nodes I'll be manually hardcoding 1 server to sleep for a few seconds. Meanwhile, if a read comes on that node it will start the read broadcast functionality. Once the server will know the value of maximum timestamp it will start the write-back and will update its value. This will simulate the behavior when there is high latency and the server is taking time to respond. In the below results we can see that write-back has been done on the server 2 while server 1 and server 3 won't start the write-back since it would already have the updated value.

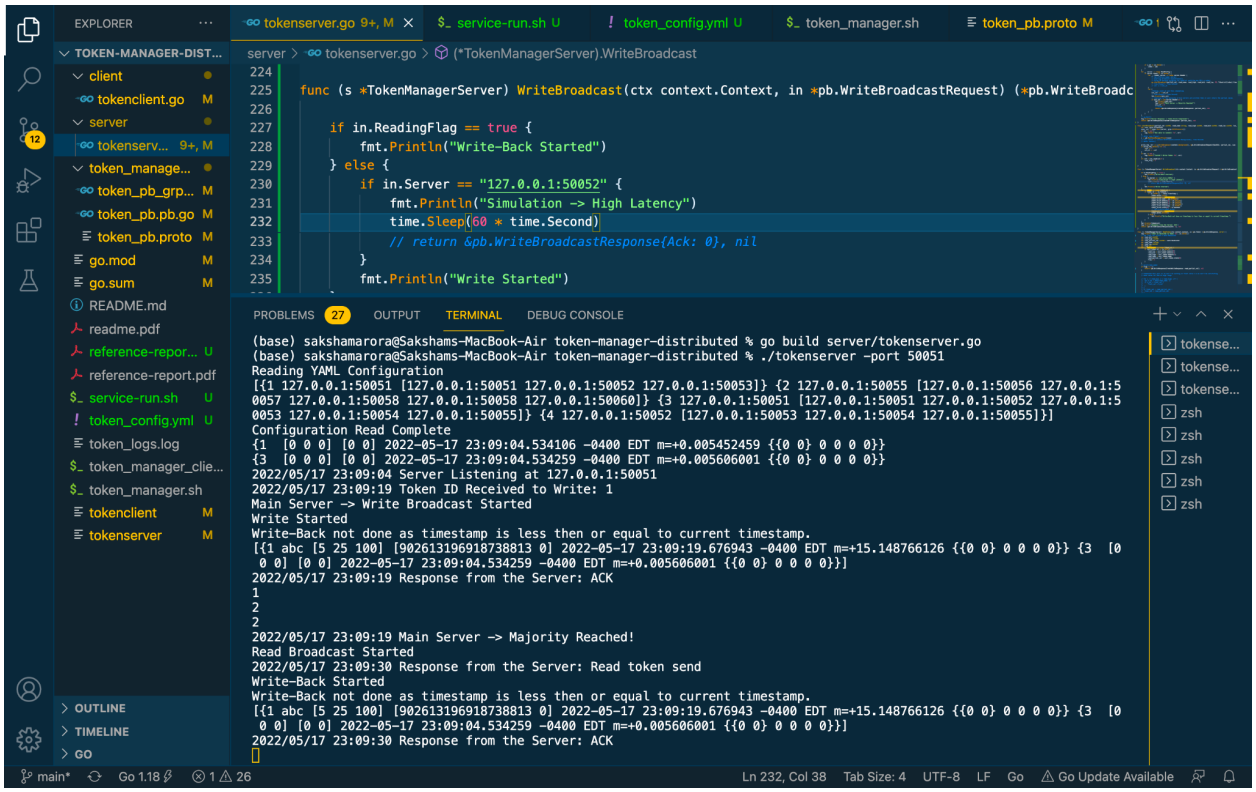
  Scenario - 1) With sleep

  Client Request)

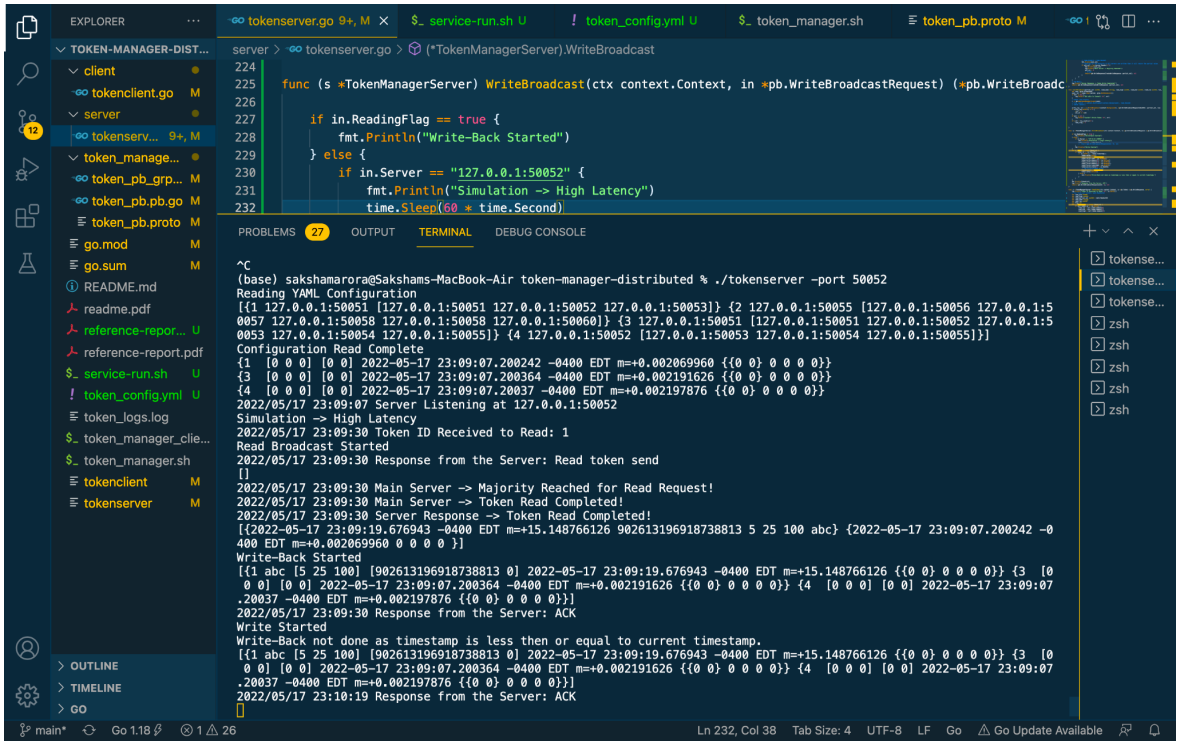
  Here we can see we have started the read request on server 127.0.0.1:50052. Additionally, we can see it's returning the correct partial value also other parameters are updated which we can see in server 2(127.0.0.1:50052) logs.



Server 127.0.0.1:50051 logs -

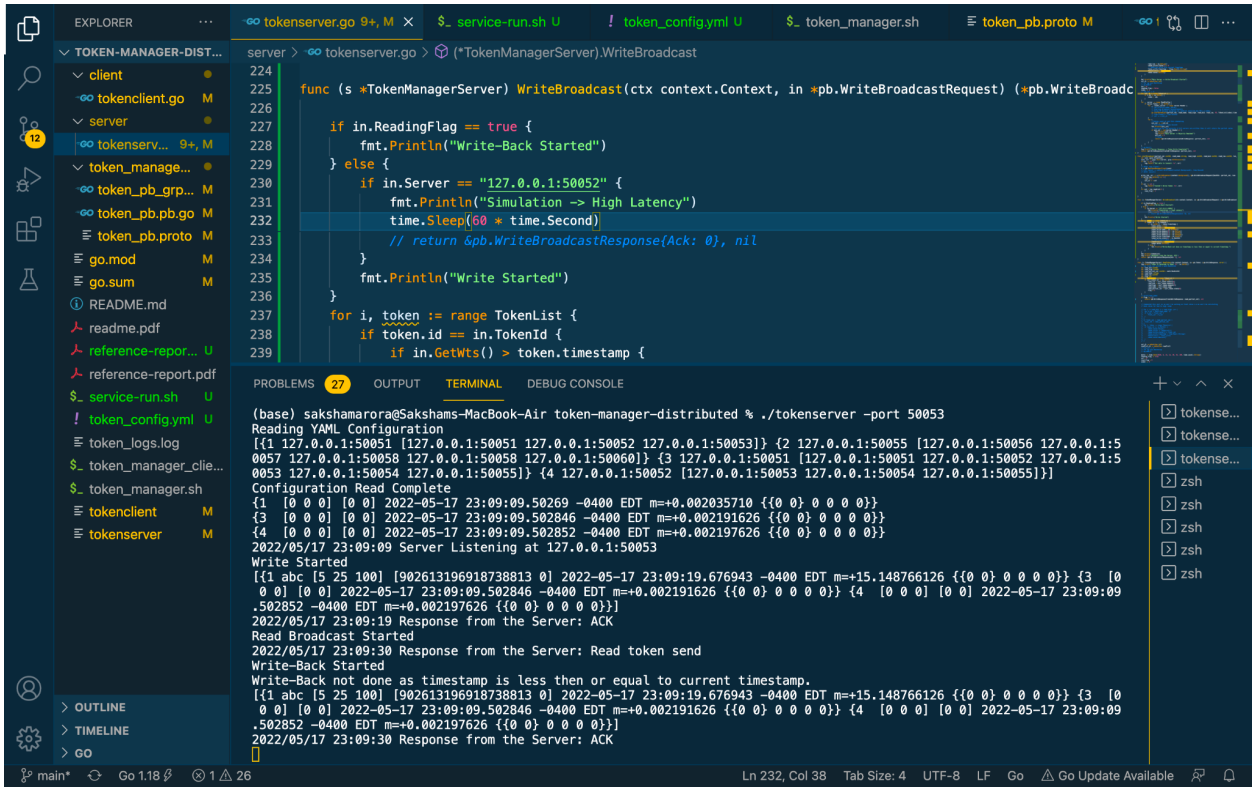


Server 127.0.0.1:50052 logs -



In the above image we can see that there was latency in that server so it was not able to write. In the meantime the read request was fired on that server and then we can see that the write-back has started. After some time, we see an older write request was trying to execute but couldn't go further as the timestamp was higher now.

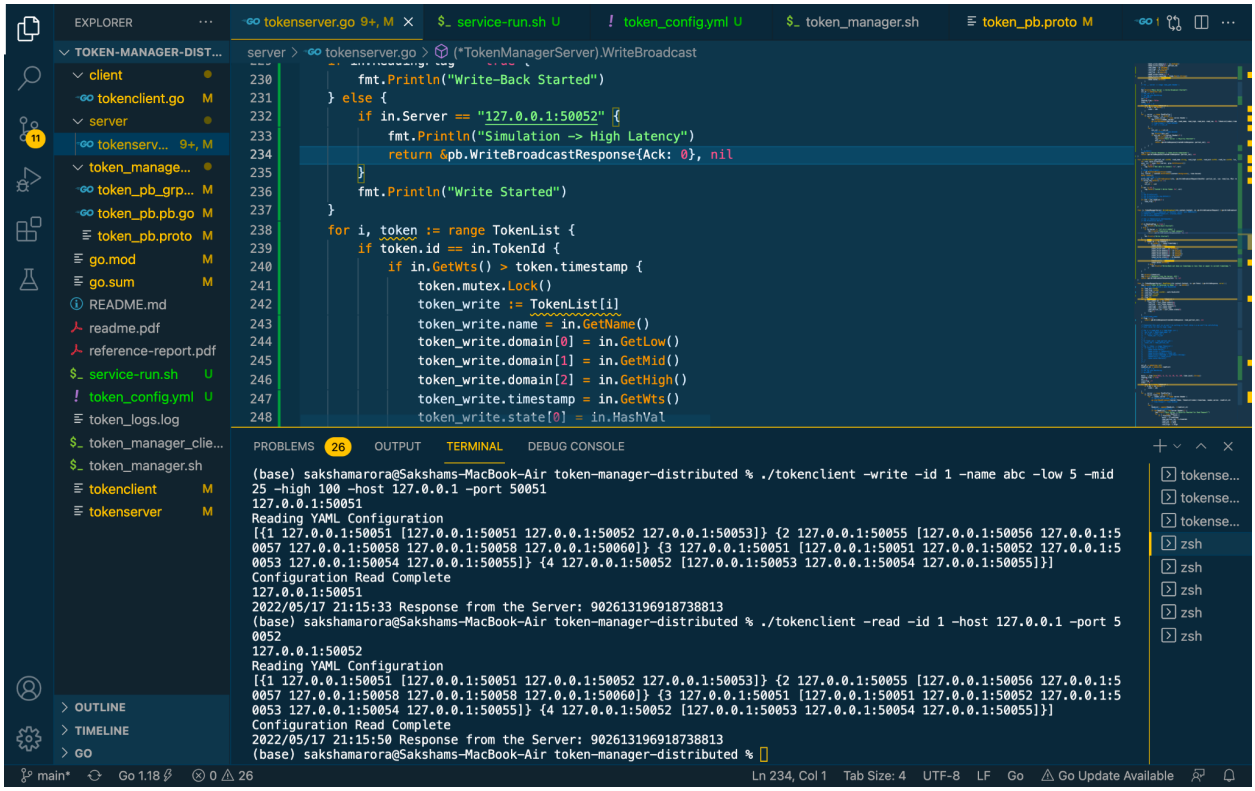
Server 127.0.0.1:50053 logs -



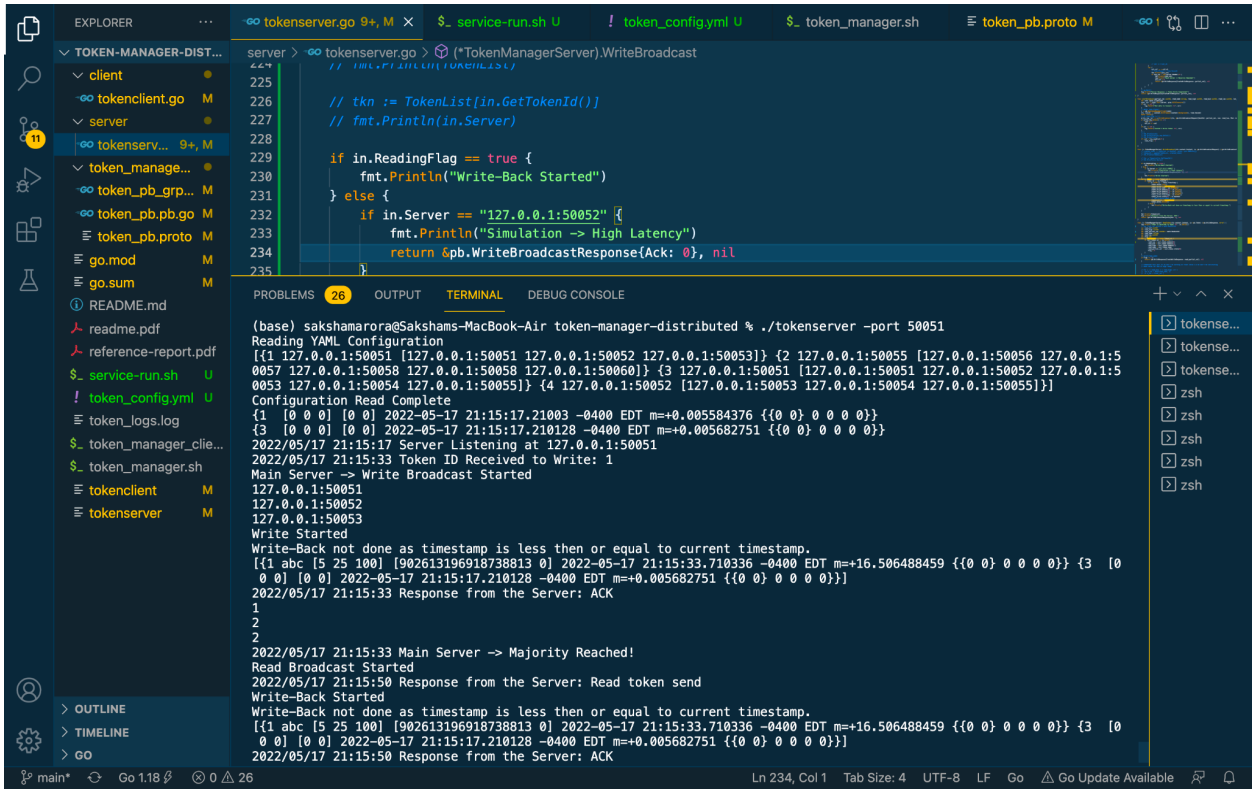
Scenario 2) No sleep. Server responds with negative acknowledgment(assume as crash)

Client Requests ->

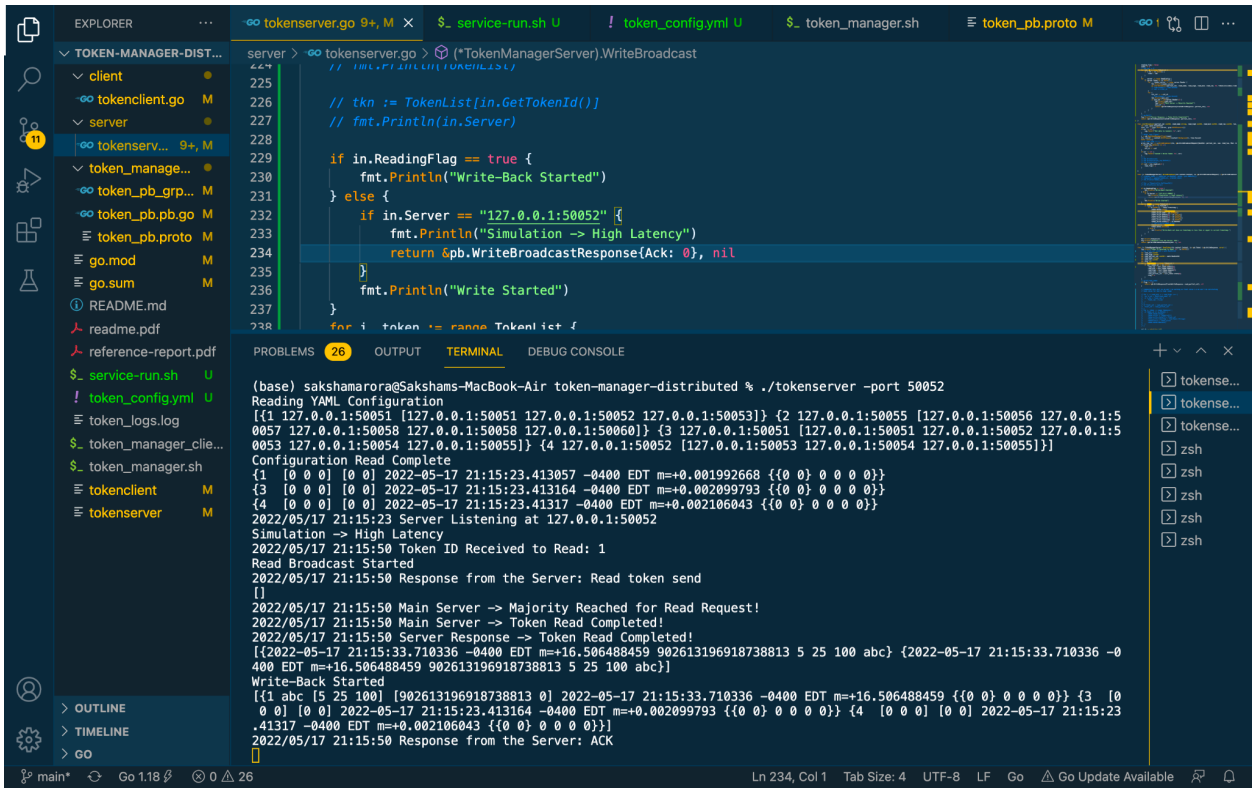
Here we can see the consistency has been achieved as server 127.0.0.1:50052 was crashed.



Server 127.0.0.1:50051 -



Server 127.0.0.1:50052 -



Above we can see the write-back started for server 2 as it was crashed. And after the write-back it maintains the value with the greater timestamp.

Server 127.0.0.1:50053 -