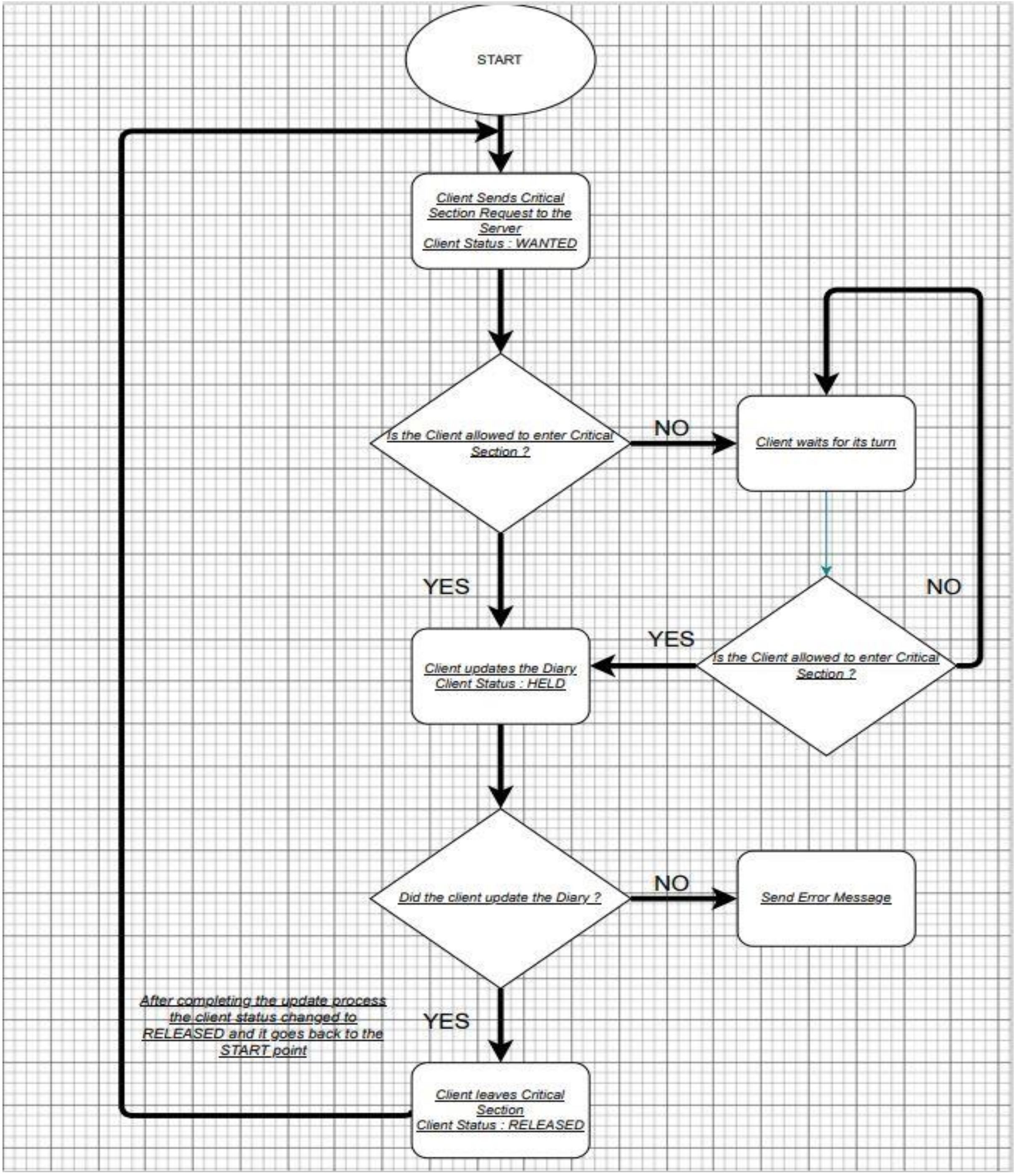
ROUCAIROL CARVALHO MUTUAL EXCLUSION IMPLEMENTATION

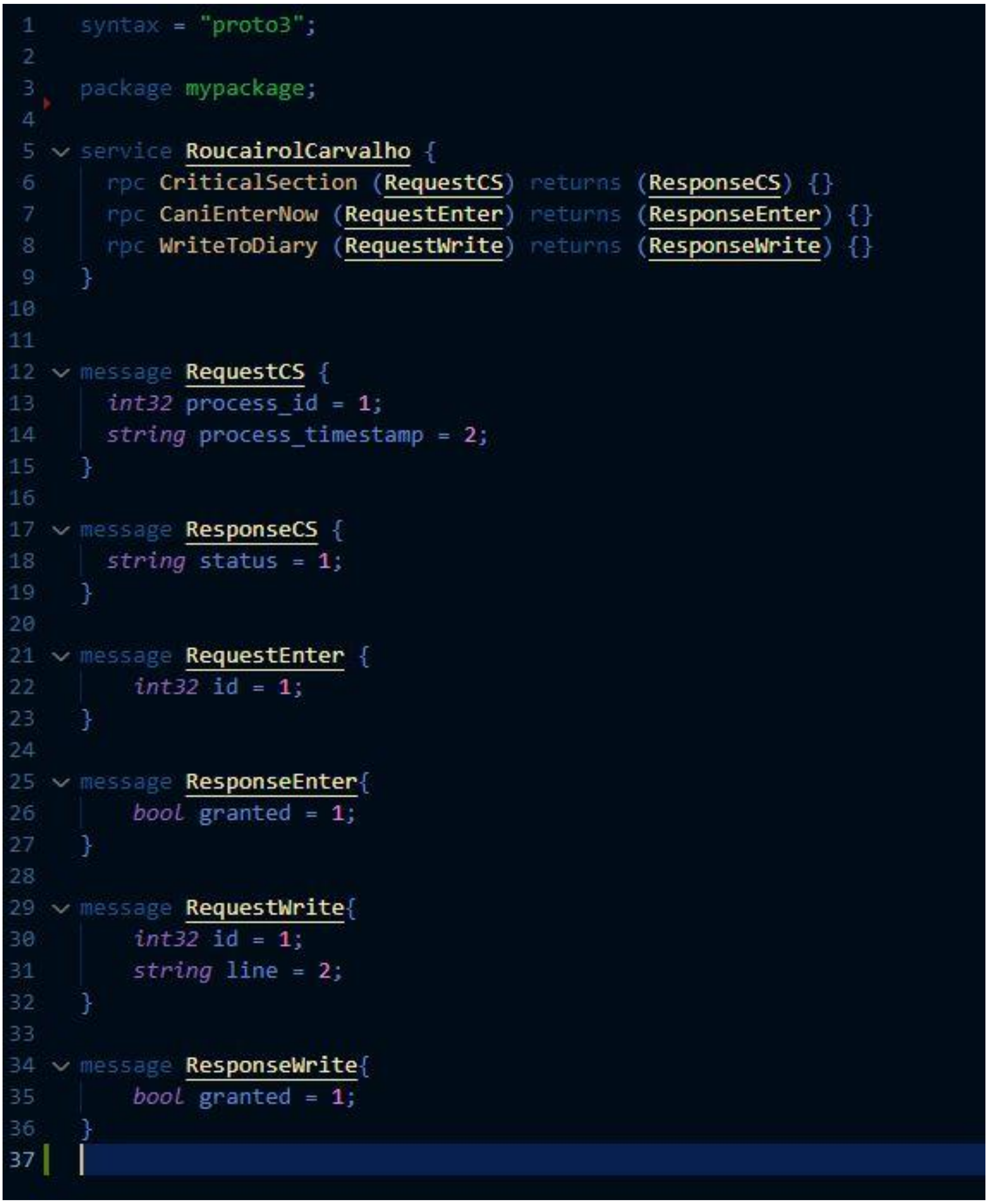
In this project, we implemented the Roucairol-Carvalho algorithm as a message update system. The clients are trying to write a message to the diary, and if they send the write request at the same time, they will be waiting to enter the critical section. The client with the highest priority will be allowed to enter the critical section to update the diary.

Below we see the UML diagram of the algorithm.


START
Client Sends Critical Section Request to the Server
Client Status : WANTED
Is the Client allowed to enter Critical Section ?
NO
Client waits for its turn
YES
NO
YES
Is the Client allowed to enter Critical Section ?
Client updates the Diary
Client Status : HELD
Did the client update the Diary ?
NO
Send Error Message
YES
After completing the update process the client status changed to RELEASED and it goes back to the START point
Client leaves Critical Section
Client Status : RELEASED

PROTOCOL BUFFERS

In this project, clients are passing information to each other using Protocol Buffers in the gRPC framework. A client sends a request to the server and waits for a response from the server on whether they can enter the critical section or not. The gRPC methods can be seen in the following Protocol Buffers snippet.

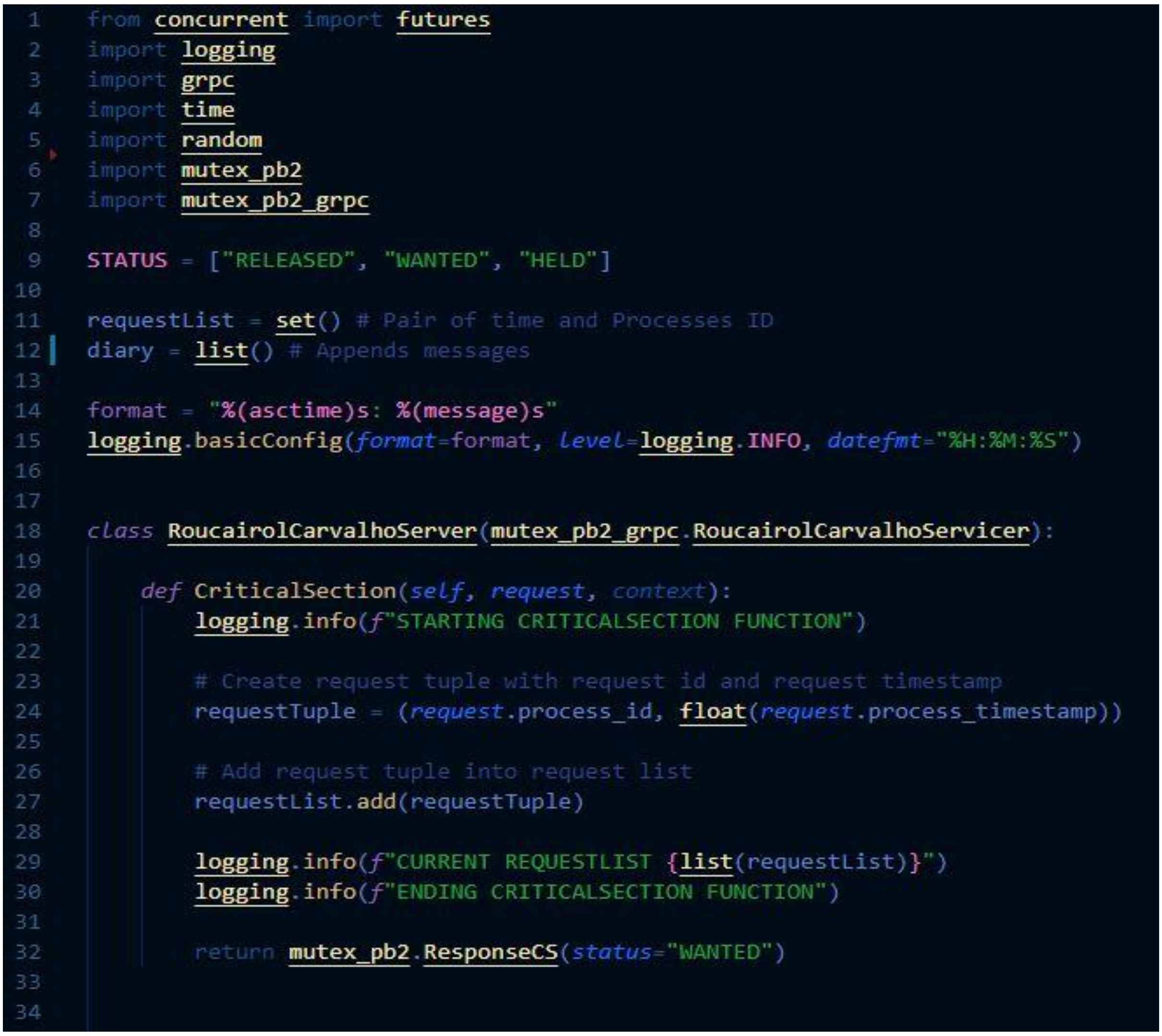


```
syntax = "proto3";

package mypackage;

service RoucairolCarvalho {
  rpc CriticalSection (RequestCS) returns (ResponseCS) {}
  rpc CaniEnterNow (RequestEnter) returns (ResponseEnter) {}
  rpc WriteToDiary (RequestWrite) returns (ResponseWrite) {}
}


message RequestCS {
  int32 process_id = 1;
  string process_timestamp = 2;
}

message ResponseCS {
  string status = 1;
}

message RequestEnter {
    int32 id = 1;
}

message ResponseEnter{
    bool granted = 1;
}

message RequestWrite{
    int32 id = 1;
    string line = 2;
}

message ResponseWrite{
    bool granted = 1;
}

```

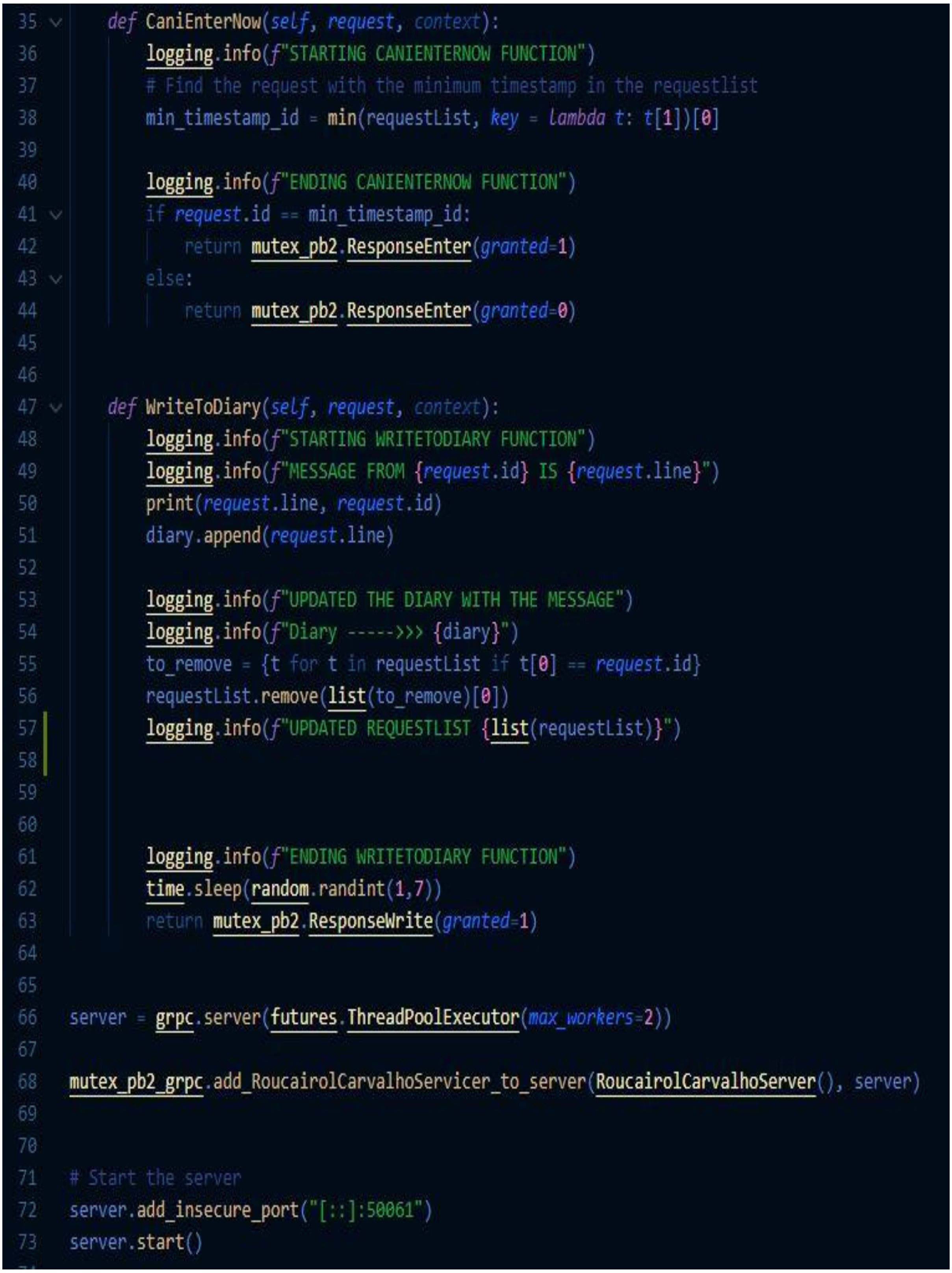
The service has three methods: CriticalSection, CanIEnterNow, and WriteToDiary. The CriticalSection method is used to send a request when a client wants to enter the critical section. At this stage, the status of the client is "WANTED". The server receives this request and checks if any other client is trying to enter the critical section. If not, the client sends another request asking if it can enter the critical section now. If the answer is yes, the client is allowed to update the diary. At this stage, the client's status is "HELD". Finally, the client sends the WriteToDiary request and, when the write procedure is complete, the client leaves the critical section and changes its status to "RELEASED".

SERVER CODE

On the server side, we override the gRPC class "RoucairolCarvalhoServer" and implement the methods for request handling. We can see the server-side code below.



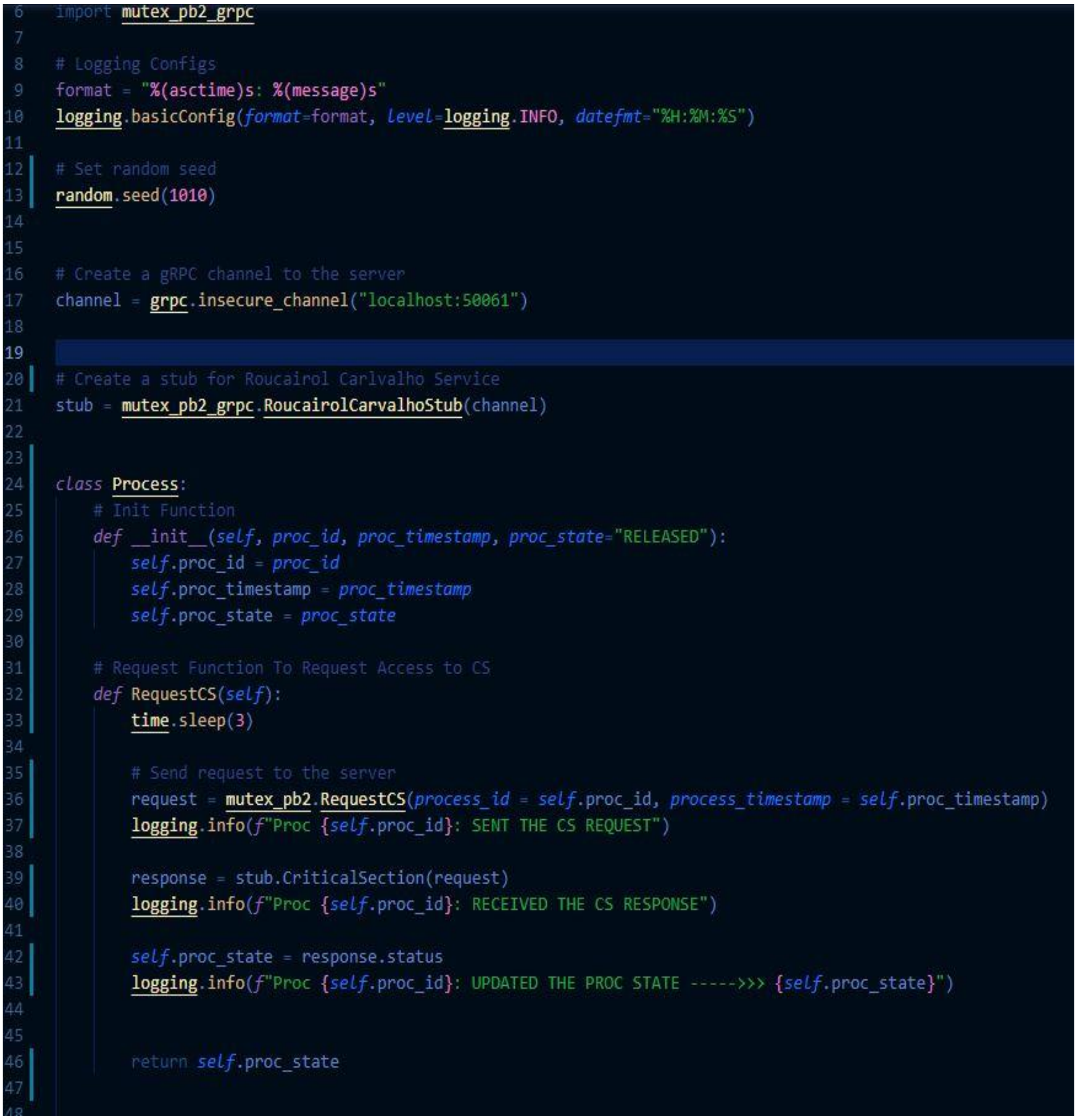
```python
from concurrent import futures
import logging
import grpc
import time
import random
import mutex_pb2
import mutex_pb2_grpc

STATUS = ["RELEASED", "WANTED", "HELD"]

requestList = set() # Pair of time and Processes ID
diary = list() # Appends messages

format = "%(asctime)s: %(message)s"
logging.basicConfig(format=format, level=logging.INFO, datefmt="%H:%M:%S")


class RoucairolCarvalhoServer(mutex_pb2_grpc.RoucairolCarvalhoServicer):

    def CriticalSection(self, request, context):
        logging.info(f"STARTING CRITICALSECTION FUNCTION")

        # Create request tuple with request id and request timestamp
        requestTuple = (request.process_id, float(request.process_timestamp))

        # Add request tuple into request list
        requestList.add(requestTuple)

        logging.info(f"CURRENT REQUESTLIST {list(requestList)}")
        logging.info(f"ENDING CRITICALSECTION FUNCTION")

        return mutex_pb2.ResponseCS(status="WANTED")


```



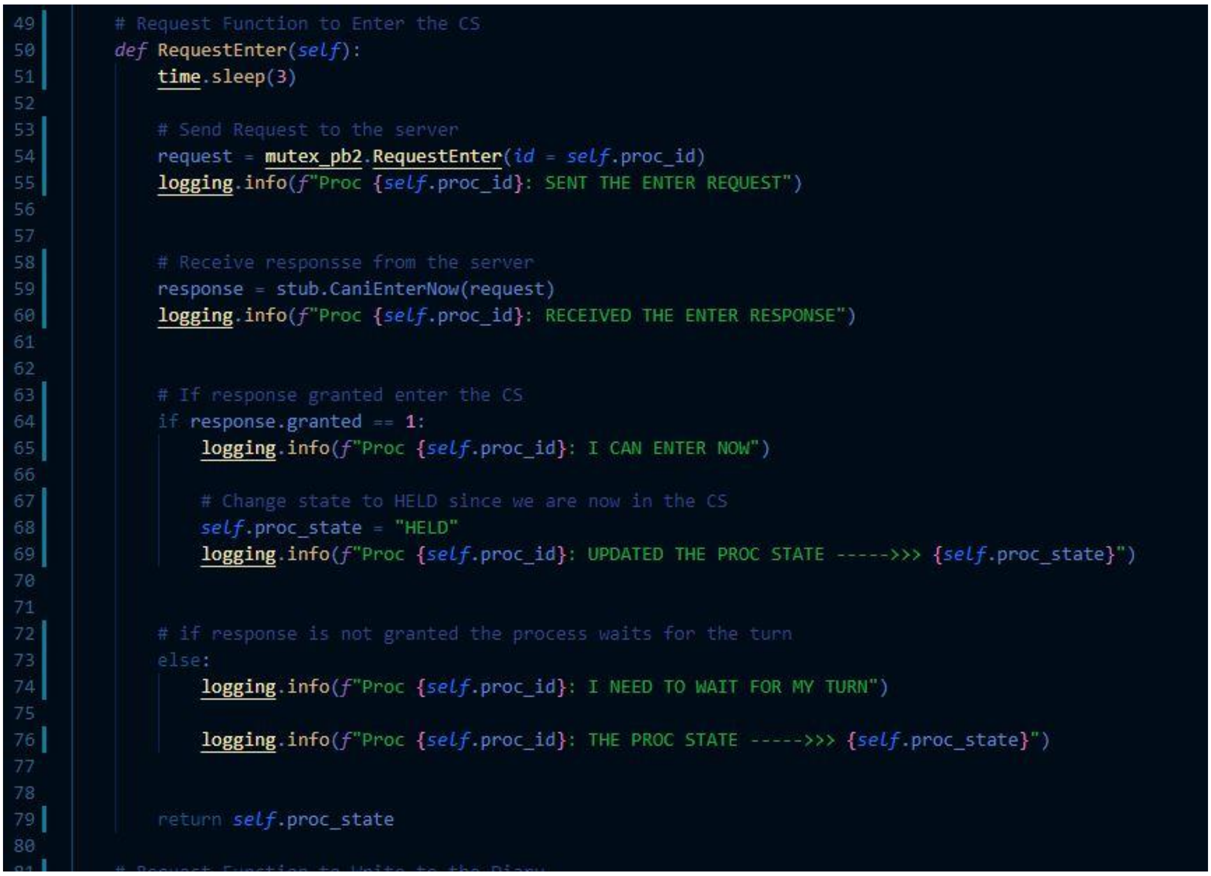
```python
    def CaniEnterNow(self, request, context):
        logging.info(f"STARTING CANIENTERNOW FUNCTION")
        # Find the request with the minimum timestamp in the requestlist
        min_timestamp_id = min(requestList, key = lambda t: t[1])[0]

        logging.info(f"ENDING CANIENTERNOW FUNCTION")
        if request.id == min_timestamp_id:
            return mutex_pb2.ResponseEnter(granted=1)
        else:
            return mutex_pb2.ResponseEnter(granted=0)


    def WriteToDiary(self, request, context):
        logging.info(f"STARTING WRITETODIARY FUNCTION")
        logging.info(f"MESSAGE FROM {request.id} IS {request.line}")
        print(request.line, request.id)
        diary.append(request.line)

        logging.info(f"UPDATED THE DIARY WITH THE MESSAGE")
        logging.info(f"Diary ----->>> {diary}")
        to_remove = {t for t in requestList if t[0] == request.id}
        requestList.remove(list(to_remove)[0])
        logging.info(f"UPDATED REQUESTLIST {list(requestList)}")



        logging.info(f"ENDING WRITETODIARY FUNCTION")
        time.sleep(random.randint(1,7))
        return mutex_pb2.ResponseWrite(granted=1)


server = grpc.server(futures.ThreadPoolExecutor(max_workers=2))

mutex_pb2_grpc.add_RoucairolCarvalhoServicer_to_server(RoucairolCarvalhoServer(), server)


# Start the server
server.add_insecure_port("[::]:50061")
server.start()
```

CLIENT CODE

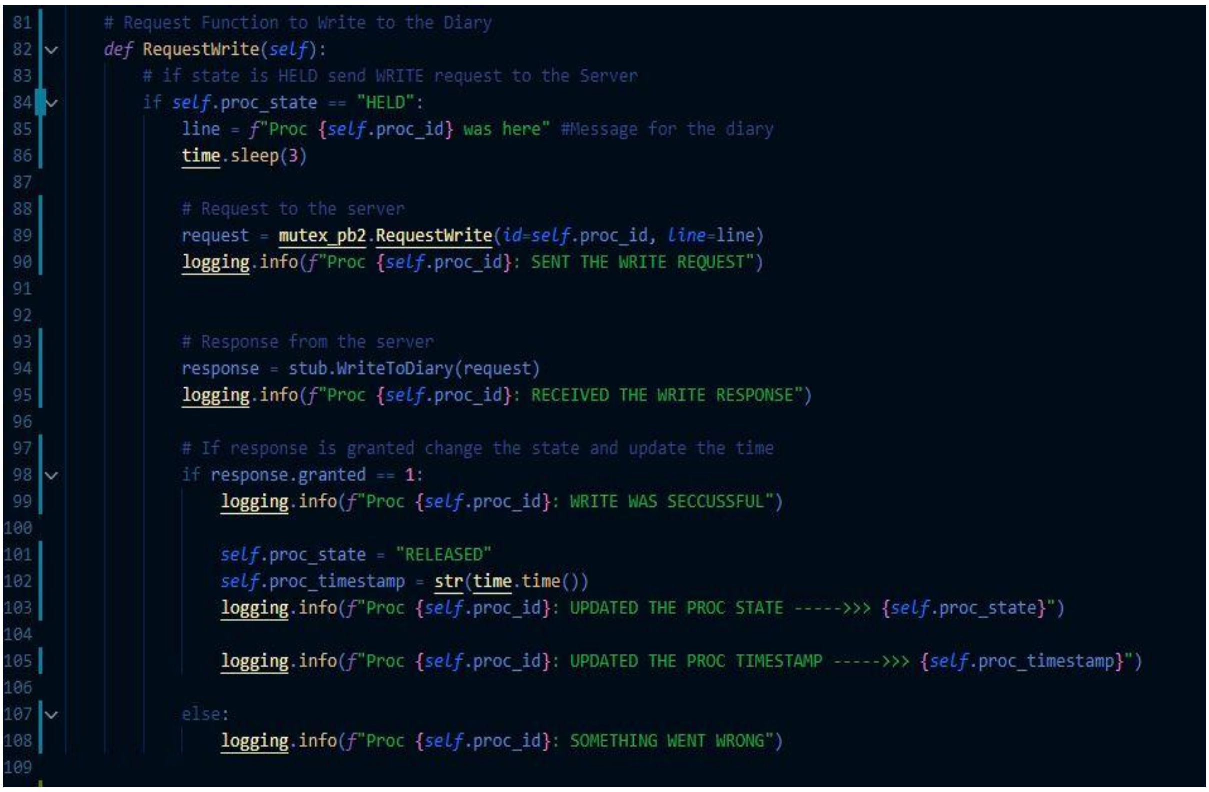
The client code contains the Process class, which we can use to instantiate a process unit and call methods on it to send requests to the server. When we call the Process class, it creates an instance that has three attributes: process_id, process_timestamp, and process_status. Also, the class has three methods: RequestCS, RequestEnter, and RequestWrite. When we run the client code, the instance is created and randomly requests to enter the critical section. Other clients can also request to enter the critical section. We can see the client-side code in the below snippets.



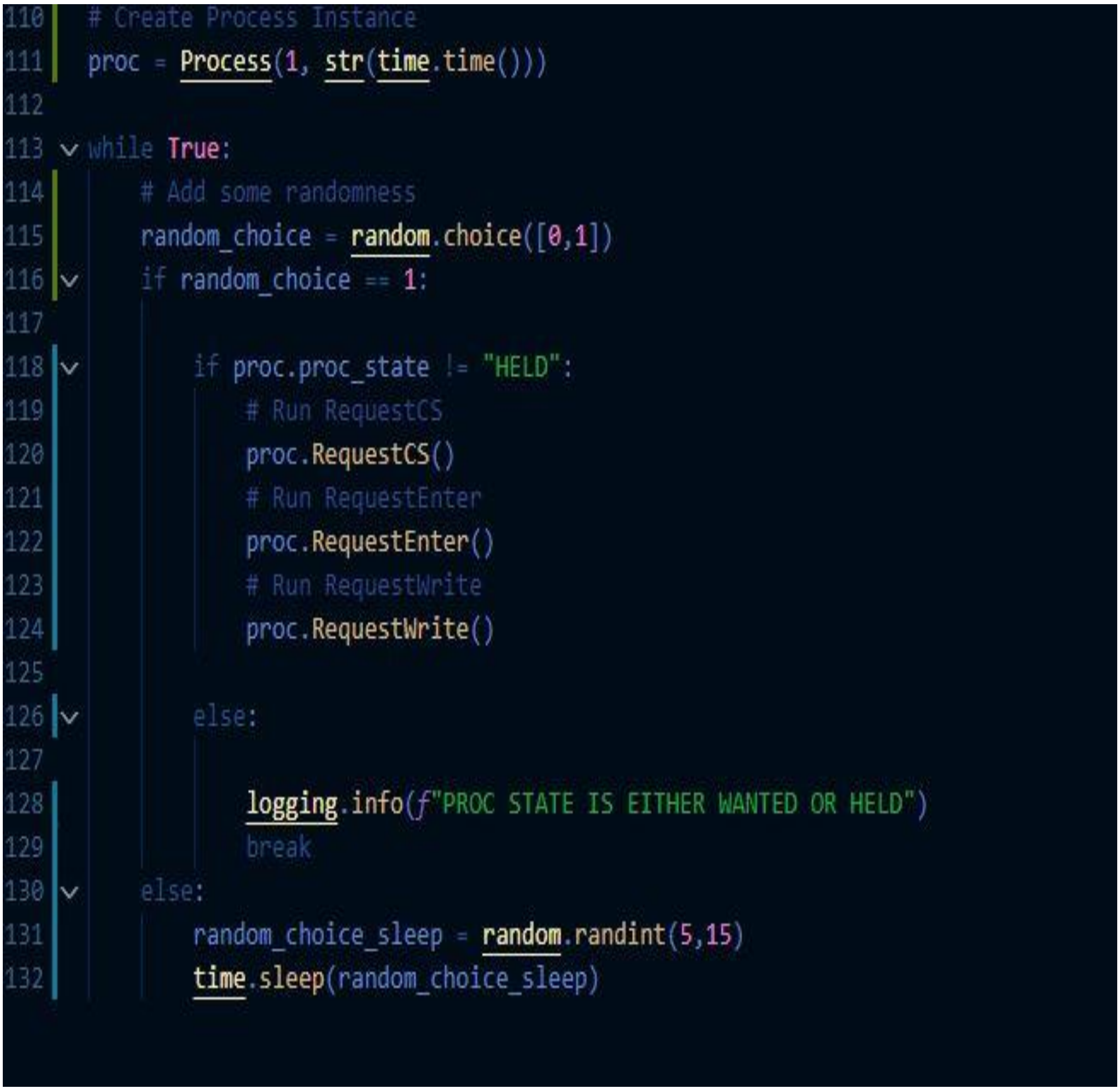
```python
import mutex_pb2_grpc

# Logging Configs
format = "%(asctime)s: %(message)s"
logging.basicConfig(format=format, level=logging.INFO, datefmt="%H:%M:%S")

# Set random seed
random.seed(1010)


# Create a gRPC channel to the server
channel = grpc.insecure_channel("localhost:50061")


# Create a stub for Roucairol Carlvalho Service
stub = mutex_pb2_grpc.RoucairolCarvalhoStub(channel)


class Process:
    # Init Function
    def __init__(self, proc_id, proc_timestamp, proc_state="RELEASED"):
        self.proc_id = proc_id
        self.proc_timestamp = proc_timestamp
        self.proc_state = proc_state

    # Request Function To Request Access to CS
    def RequestCS(self):
        time.sleep(3)

        # Send request to the server
        request = mutex_pb2.RequestCS(process_id = self.proc_id, process_timestamp = self.proc_timestamp)
        logging.info(f"Proc {self.proc_id}: SENT THE CS REQUEST")

        response = stub.CriticalSection(request)
        logging.info(f"Proc {self.proc_id}: RECEIVED THE CS RESPONSE")

        self.proc_state = response.status
        logging.info(f"Proc {self.proc_id}: UPDATED THE PROC STATE ----->>> {self.proc_state}")


        return self.proc_state

```



```
    # Request Function to Enter the CS
    def RequestEnter(self):
        time.sleep(3)

        # Send Request to the server
        request = mutex_pb2.RequestEnter(id = self.proc_id)
        logging.info(f"Proc {self.proc_id}: SENT THE ENTER REQUEST")


        # Receive responsse from the server
        response = stub.CaniEnterNow(request)
        logging.info(f"Proc {self.proc_id}: RECEIVED THE ENTER RESPONSE")


        # If response granted enter the CS
        if response.granted == 1:
            logging.info(f"Proc {self.proc_id}: I CAN ENTER NOW")

            # Change state to HELD since we are now in the CS
            self.proc_state = "HELD"
            logging.info(f"Proc {self.proc_id}: UPDATED THE PROC STATE ----->>> {self.proc_state}")


        # if response is not granted the process waits for the turn
        else:
            logging.info(f"Proc {self.proc_id}: I NEED TO WAIT FOR MY TURN")

            logging.info(f"Proc {self.proc_id}: THE PROC STATE ----->>> {self.proc_state}")


        return self.proc_state
```



```
    # Request Function to Write to the Diary
    def RequestWrite(self):
        # if state is HELD send WRITE request to the Server
        if self.proc_state == "HELD":
            line = f"Proc {self.proc_id} was here" #Message for the diary
            time.sleep(3)

            # Request to the server
            request = mutex_pb2.RequestWrite(id=self.proc_id, line=line)
            logging.info(f"Proc {self.proc_id}: SENT THE WRITE REQUEST")


            # Response from the server
            response = stub.WriteToDiary(request)
            logging.info(f"Proc {self.proc_id}: RECEIVED THE WRITE RESPONSE")

            # If response is granted change the state and update the time
            if response.granted == 1:
                logging.info(f"Proc {self.proc_id}: WRITE WAS SECCUSSFUL")

                self.proc_state = "RELEASED"
                self.proc_timestamp = str(time.time())
                logging.info(f"Proc {self.proc_id}: UPDATED THE PROC STATE ----->>> {self.proc_state}")

                logging.info(f"Proc {self.proc_id}: UPDATED THE PROC TIMESTAMP ----->>> {self.proc_timestamp}")

            else:
                logging.info(f"Proc {self.proc_id}: SOMETHING WENT WRONG")
```



```python
# Create Process Instance
proc = Process(1, str(time.time()))

while True:
    # Add some randomness
    random_choice = random.choice([0,1])
    if random_choice == 1:

        if proc.proc_state != "HELD":
            # Run RequestCS
            proc.RequestCS()
            # Run RequestEnter
            proc.RequestEnter()
            # Run RequestWrite
            proc.RequestWrite()

        else:

            logging.info(f"PROC STATE IS EITHER WANTED OR HELD")
            break
    else:
        random_choice_sleep = random.randint(5,15)
        time.sleep(random_choice_sleep)
```

The client code runs constantly and always sends requests to the server. However, we added some randomness to the server code to sometimes skip requests.