50.041 Distributed Systems and Computing

Final Report
Sep-Dec 2024

Group 5

1006211	Guruprasath Gopal
1005922	Koh Chee Kiat
1006591	Rajavelu Sree Devi
1006111	Nithyashree D/O Ramanathan Vairavan
1005985	Noven Zen Hong Guan

GitHub Repo: Main Branch - Scalability Testing Branch

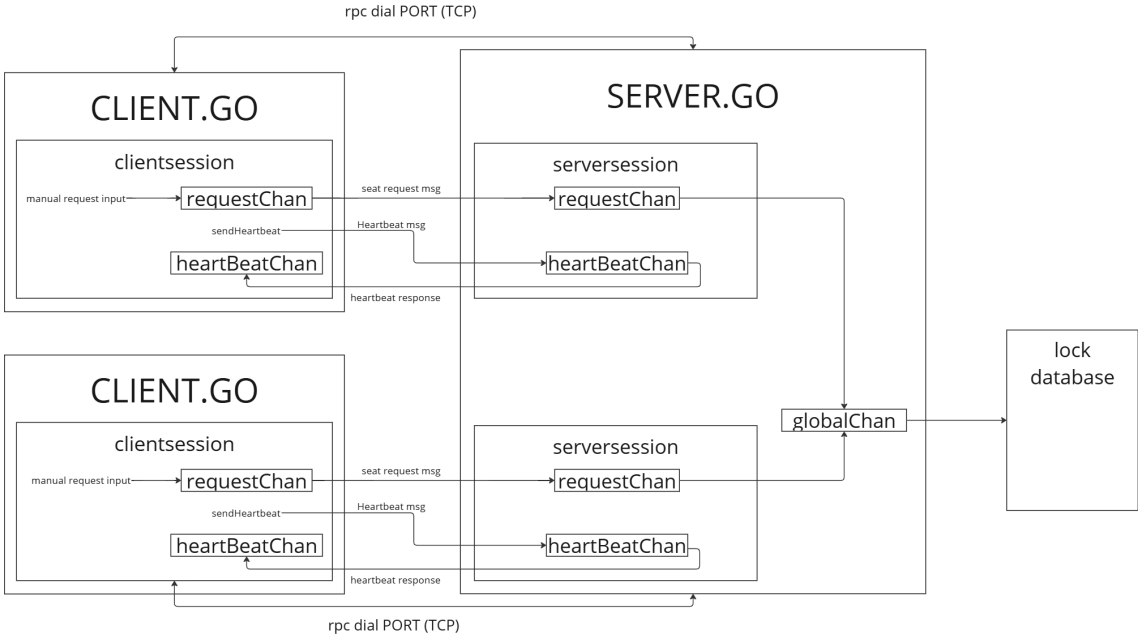
Introduction

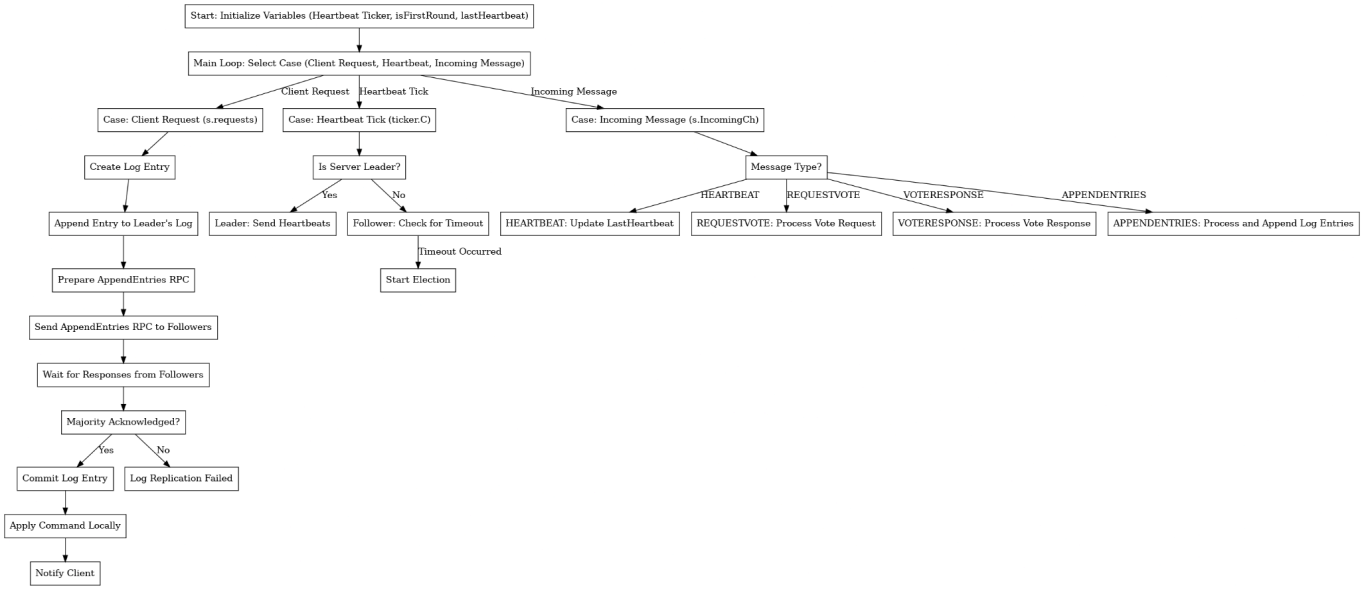
Project scope

Our project is a movie booking system designed to provide users with a seamless and reliable experience reserving and managing their seats. Users can reserve new seats or cancel seats that they had previously reserved through interaction with the Command Line Interface (CLI). This system ensures that availability is accurately maintained in real-time, preventing double bookings of seats and potential conflicts. Our system is resilient to server faults and ensures consistency in the presence of faults.

System Architecture

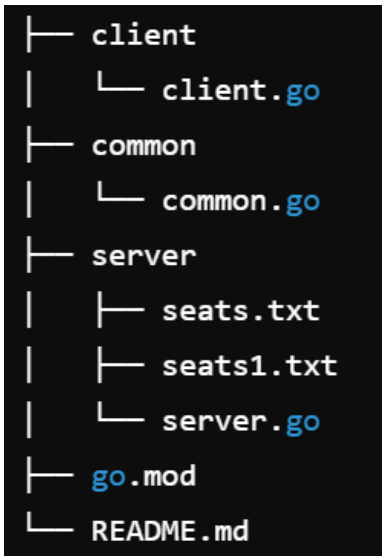
Architecture example of multiple client to one server:





File Directory and Organisation

We have separated the relevant Golang structs, constants and functions within the respective folders for ‘client’ and ‘server’ while placing structs which are common in both server and client implementations in the folder ‘common’. Instructions on how to run and interact with the program are mentioned in the main branch’s README.md.



Client

Each client is simulated via Client.go. When started, Client.go instantiates a client session and prompts Server.go to instantiate a server session to communicate with the client session. Therefore, the server and client sessions have a one-to-one relationship.

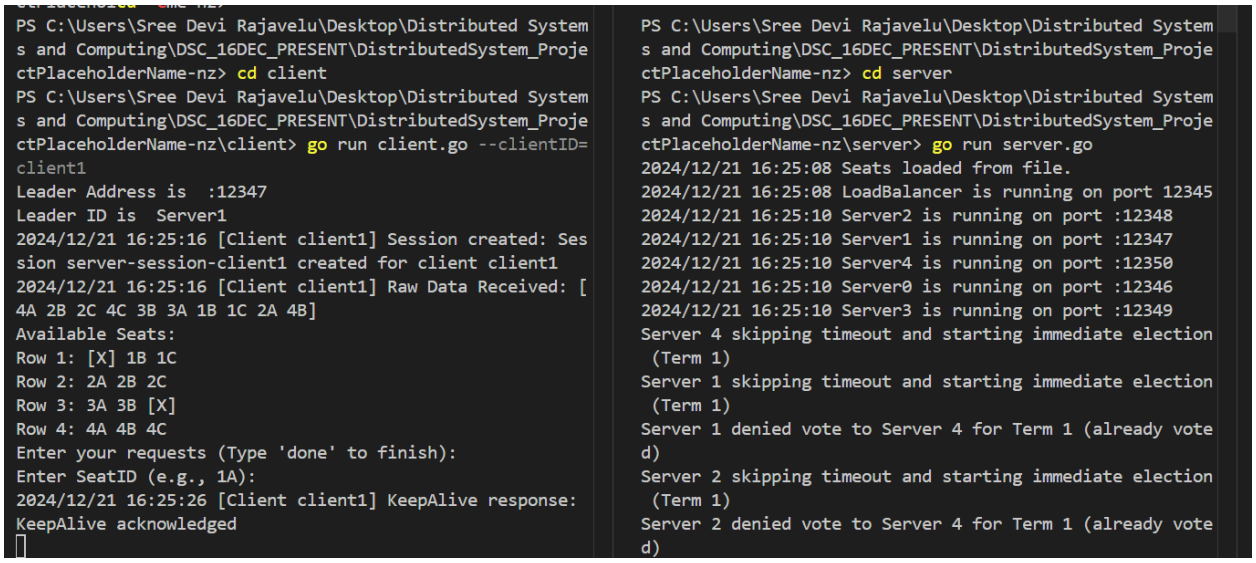
Each client session has a request channel to store and process all manually input seat requests, and a heartbeat channel to receive heartbeat responses from the server session.

Server

The Chubby server is simulated via Server.go. The server instantiates a server session for each client session. Each server session has a request channel to receive client session requests and a heartbeat channel to receive heartbeat messages from client sessions.

The server session sends all request messages to the server's global channel to be processed. A single global queue ensures atomicity for request processing.

Demonstration of starting and interacting with the application:



Features

Client Main Features

1. Session Management
    a. Manages its session with the server by creating, maintaining, and terminating the session.
    b. Ensures that the client can make requests during the session and gracefully shut down when the session ends.
2. Keep-Alive Mechanism
    a. The client sends period heartbeat (KEEPALIVE) messages to the server to ensure that the connection remains active and to check the server's functionality.
    b. If no response is received from the server, the client assumes that the master server is down.
3. Error Handling
    a. Handles errors such as failure to send requests or failure to connect to a server.
    b. Logs messages for debugging and ensure the errors do not crash the client.
4. Usage CLI
    a. The project makes use of the CLI to take in user inputs such as seat id (1A,1B..) and request type (RESERVE or CANCEL).
5. Request Handling
    a. The individual inputs from the CLI are parsed into a request struct and the request is added to the request channel.

Server Main Features

Session Management

1. Session Management
    a. Manages the session with the client by creating, maintaining, and terminating the session.
2. Request Processing
    a. The leader server processes requests and determines whether they should be served by checking the persistent storage and checking that a majority of the follower servers have appended the log entry if a request is to be served.
3. Keep-Alive Mechanism
    a. The servers send periodic heartbeat messages to each other to ensure that the connection is active and detect if the leader server is down.
    b. If no heartbeat message is received from a server, it is assumed to be down and follower servers start an election.

4. Error Handling
    a. Check for the validity of the request in the request channel by checking if the seat ID exists and if the request type (RESERVE or CANCEL) input by a client is valid.
    b. Error in loading text file.
    c. Error in updating changed seat status to the text file.
5. Seat Management
    a. Loads seat availability and status from the text file for usage.
    b. Updates are made to the text file when there are changes to a seat status.
    c. Ensures that a reserved seat cannot be reserved by another client again.
    d. Ensures that a client can only cancel a reservation to a seat if the client is the client that reserved the seat.
6. Consistency
    a. Log replication is initiated by the leader server (ReplicateLog function) for all follower servers when it processes a request from a client (ProcessRequest function).
    b. When any follower server receives an 'APPENDENTRIES' message from the leader server, it checks the log index and term of the incoming log entry, comparing it with its highest log entry index and term (handleAppendEntries function).
    c. Only if the incoming log entry comes with a previous log index and term that matches or is more updated than the receiving follower server's log with the highest index, the log entry is appended to the follower server's log.
    d. With a log entry appended successfully to the follower server's log, the follower sends a 'APPENDENTRIESRESPONSE' message which includes the follower's term and a success status (boolean to indicate whether the log entry was appended).
    e. Upon receiving successful responses from a majority of the follower servers, the leader server will first check that the commit index is greater than the last applied commit index to prevent outdated entries from being committed (applyLogEntries function). The leader then commits the log entries by saving the updated state (seats) to the persistent storage (seats.txt).

7. Fault Tolerance
    a. Log Consistency:
        i. Followers reject AppendEntries RPCs with mismatched logs, preventing corruption.
        ii. Followers can truncate conflicting log entries and synchronise with the leader, ensuring eventual consistency.
    b. Majority-Based Commit:

i. A log entry is considered committed only when a majority of servers (quorum) replicate it. This ensures durability despite minority server failures.

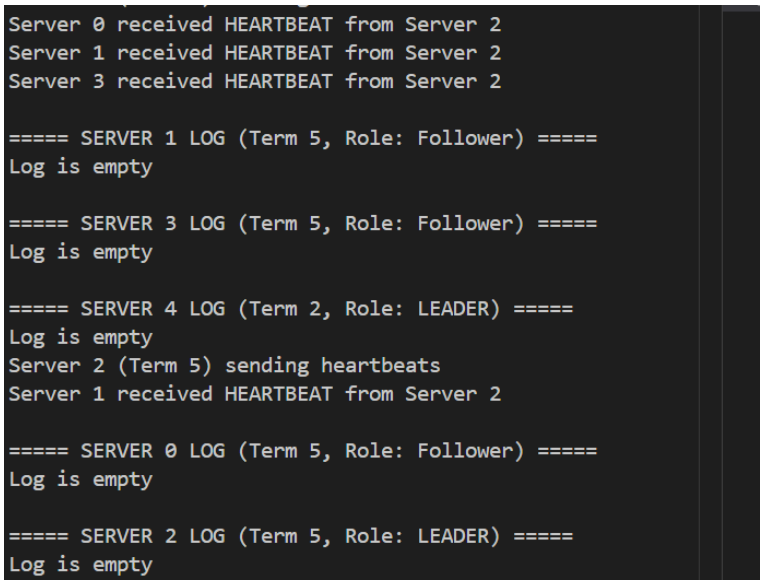
8. Simulating leader server failure (please refer to instructions on how to simulate leader failure in the main branch's README.md) :

```
===== SERVER 0 LOG (Term 2, Role: Follower) =====
Log is empty

===== SERVER 4 LOG (Term 2, Role: LEADER) =====
Log is empty
Server 4 (Term 2) sending heartbeats
Server 3 received HEARTBEAT from Server 4

===== SERVER 2 LOG (Term 2, Role: Follower) =====
Log is empty

===== SERVER 1 LOG (Term 2, Role: Follower) =====
Log is empty

===== SERVER 3 LOG (Term 2, Role: Follower) =====
Log is empty
Server 2 received HEARTBEAT from Server 4

**************************Simulating Leader Failure****
**********************
Server 0 received HEARTBEAT from Server 4
Server 1 received HEARTBEAT from Server 4
servertimeout + localDelay is 4.406474776s
2024/12/15 20:41:25 Server 1: No heartbeat received. Star
ting election (Term 2 -> Term 3)
servertimeout + localDelay is 4.260415683s
2024/12/15 20:41:25 Server 3: No heartbeat received. Star
ting election (Term 2 -> Term 3)
servertimeout + localDelay is 3.724925419s
```

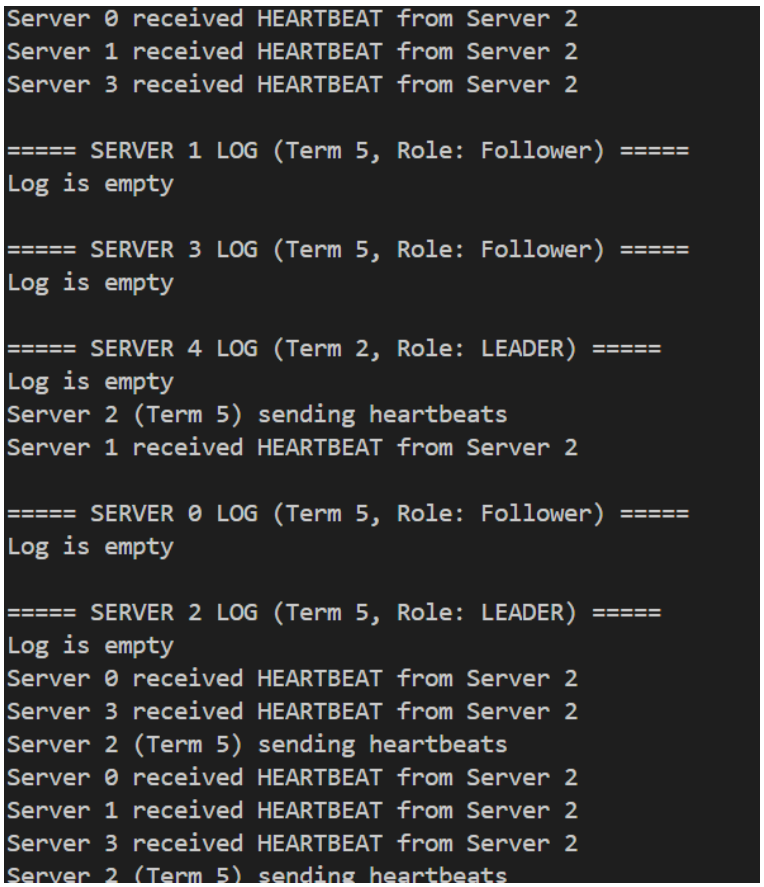
a. Number of votes received by each candidate during the election:

```
===== SERVER 3 LOG (Term 4, Role: Candidate) =====
Log is empty
Server 1 granted vote to Server 2 for Term 5
Server 2 received vote. Total votes: 2
Server 3 granted vote to Server 2 for Term 5
Server 2 received vote. Total votes: 3
Server 2 (Term 5) sending heartbeats
Server 2 became Leader for Term 5
Updating LoadBalancer with new leader info: :12348, Serve
r2
```

b. Updated role of each server in Term 5 (latest term):

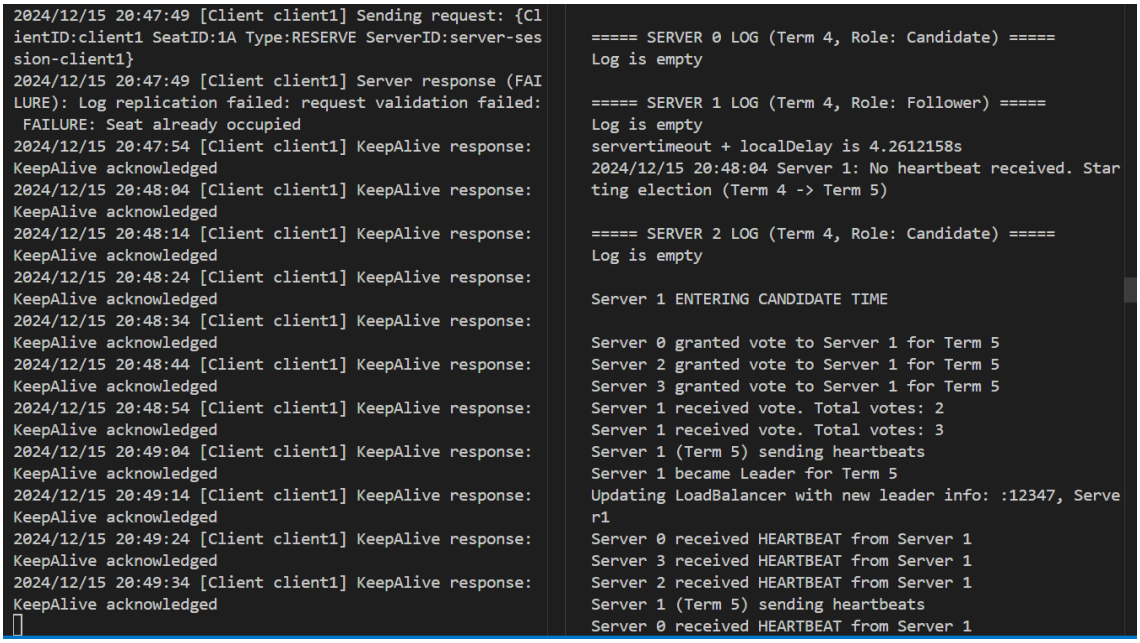


```
Server 0 received HEARTBEAT from Server 2
Server 1 received HEARTBEAT from Server 2
Server 3 received HEARTBEAT from Server 2

===== SERVER 1 LOG (Term 5, Role: Follower) =====
Log is empty

===== SERVER 3 LOG (Term 5, Role: Follower) =====
Log is empty

===== SERVER 4 LOG (Term 2, Role: LEADER) =====
Log is empty
Server 2 (Term 5) sending heartbeats
Server 1 received HEARTBEAT from Server 2

===== SERVER 0 LOG (Term 5, Role: Follower) =====
Log is empty

===== SERVER 2 LOG (Term 5, Role: LEADER) =====
Log is empty
```



```
Server 0 received HEARTBEAT from Server 2
Server 1 received HEARTBEAT from Server 2
Server 3 received HEARTBEAT from Server 2

===== SERVER 1 LOG (Term 5, Role: Follower) =====
Log is empty

===== SERVER 3 LOG (Term 5, Role: Follower) =====
Log is empty

===== SERVER 4 LOG (Term 2, Role: LEADER) =====
Log is empty
Server 2 (Term 5) sending heartbeats
Server 1 received HEARTBEAT from Server 2

===== SERVER 0 LOG (Term 5, Role: Follower) =====
Log is empty

===== SERVER 2 LOG (Term 5, Role: LEADER) =====
Log is empty
Server 0 received HEARTBEAT from Server 2
Server 3 received HEARTBEAT from Server 2
Server 2 (Term 5) sending heartbeats
Server 0 received HEARTBEAT from Server 2
Server 1 received HEARTBEAT from Server 2
Server 3 received HEARTBEAT from Server 2
Server 2 (Term 5) sending heartbeats
```

9. The Loadbalancer is updated with the newly elected server's information and the client session is kept alive during leader server failure:



Evaluation

Scalability

Note: Switch to Scalability_Testing branch (Scalability Testing Branch **)**
The code has been modified to allow multiple clients to run on the same terminal, which eases the testing process. Please follow the README.md in the Scalability_Testing branch for instructions to run the code.

We measured the average response time per request and increased the number of concurrent requests to simulate network load (randomly chosen clients made these requests) The results show a clear trend of how the system handles increasing loads.

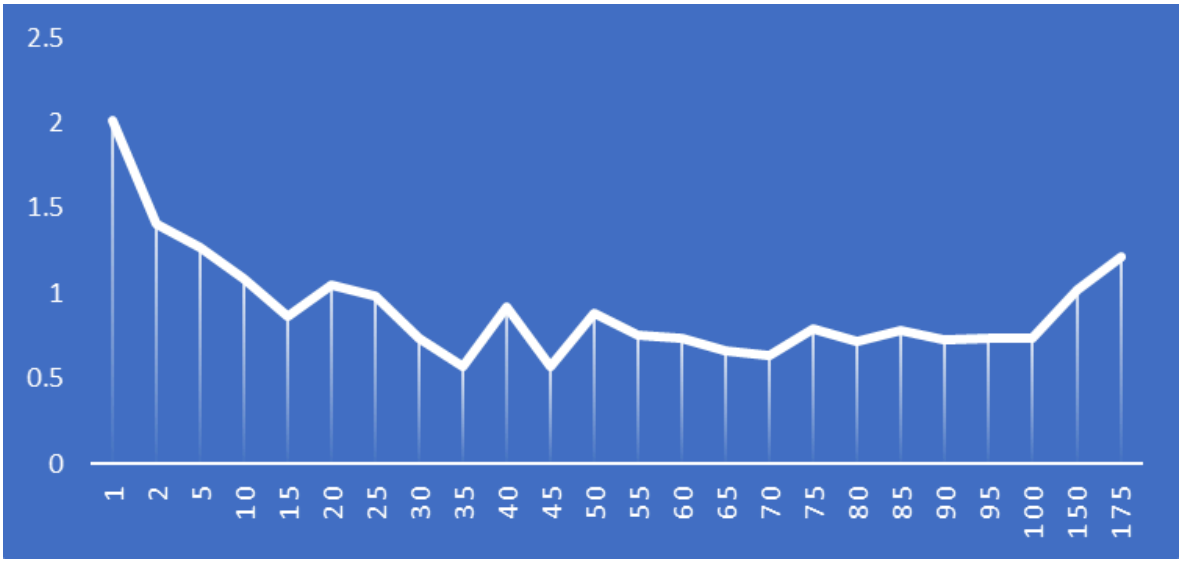
We can see that as concurrent requests increase, the average response time per request initially decreases, reaching optimal performance around 35 requests (0.57 ms). Between 30 and 70 requests, the system maintains steady performance.

However, beyond 70 concurrent requests, response times gradually rise, reaching 1.02 ms at 150 requests and 1.21 ms at 175 requests. This indicates the system approaches its performance limits under higher loads while remaining functional.

Number of Concurrent Requests	Time Taken (ms) (Averaged over 3 readings)	Average Time Taken per Request (ms)
1	2.01	2.01
2	2.82	1.41
5	6.34	1.27
10	10.83	1.08
15	12.94	0.86
20	20.92	1.05
25	24.54	0.98
30	22.13	0.74
35	19.91	0.57
40	36.80	0.92
45	25.64	0.57
50	44.02	0.88
55	41.53	0.76
60	44.08	0.73
65	42.89	0.66
70	44.14	0.63
75	59.09	0.79
80	57.75	0.72
85	66.34	0.78
90	65.60	0.73
95	70.24	0.74

100	73.54	0.74
150	152.62	1.02
175	211.69	1.21

Line Graph of Average Time Taken per request (measured in ms):



From theory, we can infer that the time complexity for our implementation would be $O(n^2)$ and as the number of concurrent requests rises, we can see an uptick in the time taken per request.

Correctness

1. Simulating the behaviour of the client reserving a seat:

```
Enter SeatID (e.g., 1A):
1A
Enter Request Type (e.g., RESERVE or CANCEL): R2024/12/15
 20:29:02 [Client client1] KeepAlive response: KeepAlive
acknowledged                                          R
ESERVE
Added request to queue: {ClientID: SeatID:1A Type:RESERVE
 ServerID:}
Enter SeatID (e.g., 1A):
2024/12/15 20:29:07 [Client client1] Sending request: {Cl
ientID:client1 SeatID:1A Type:RESERVE ServerID:server-ses
sion-client1}
2024/12/15 20:29:07 [Client client1] Server response (SUC
CESS): Operation RESERVE for seat 1A processed successful
ly
```

2. Simulating behaviour of reserving seats (server side):

```
Server 1 (Term 5) sending heartbeats
Server 4 received HEARTBEAT from Server 1
Server 2 received HEARTBEAT from Server 1
Server 3 received HEARTBEAT from Server 1
2024/12/15 20:29:02 Session server-session-client1 is in
jeopardy. Allowing grace period...
2024/12/15 20:29:02 Session server-session-client1 recove
red during grace period.
```

3. Logs of all servers before any request is processed by the leader server and receipt of 'APPENDENTRIES' by follower servers:

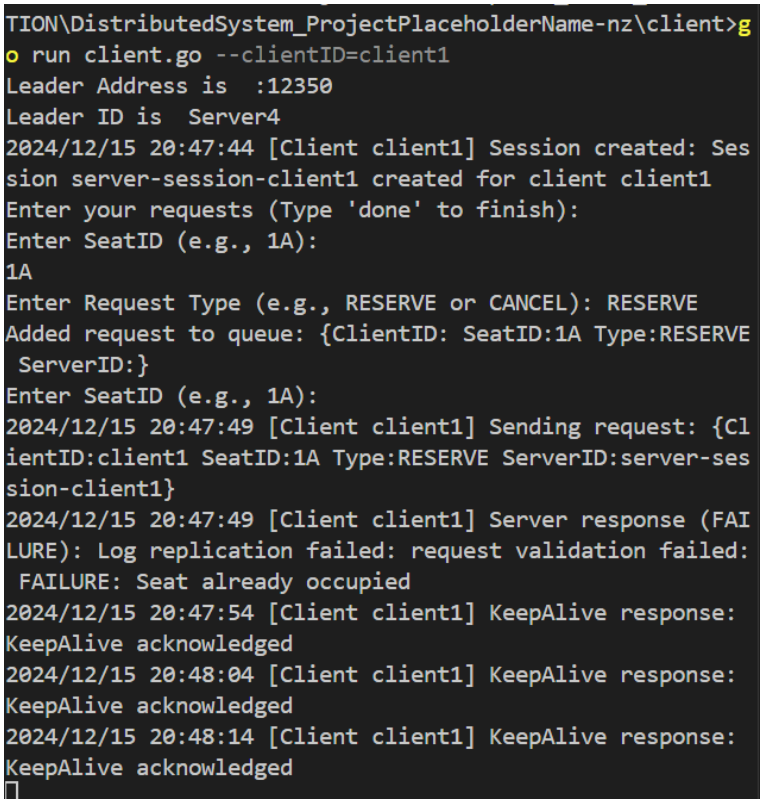
```
===== SERVER 4 LOG (Term 5, Role: Follower) =====
Log is empty
Server 1 (Term 5) sending heartbeats
Server 4 received HEARTBEAT from Server 1

===== SERVER 2 LOG (Term 5, Role: Follower) =====
Log is empty

===== SERVER 1 LOG (Term 5, Role: LEADER) =====
Log is empty
Server 2 received HEARTBEAT from Server 1
Server 3 received HEARTBEAT from Server 1

===== SERVER 3 LOG (Term 5, Role: Follower) =====
Log is empty

===== SERVER 0 LOG (Term 2, Role: LEADER) =====
Log is empty
++++++++++++++++ Server 1 (Term 5) calls replicate Log
Sending APPENDENTRIES TO ALL SERVERS
Sending APPENDENTRIES TO ALL SERVERS
Sending APPENDENTRIES TO ALL SERVERS
Sending APPENDENTRIES TO ALL SERVERS
Sending APPENDENTRIES TO ALL SERVERS
```

4. Successfully appended log entry to all follower servers:

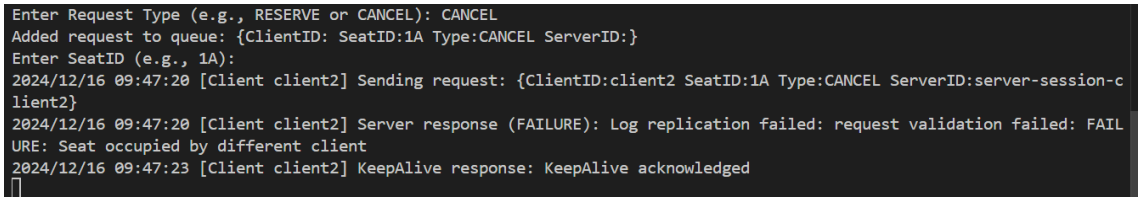
```
===== SERVER 4 LOG (Term 5, Role: Follower) =====
Index | Term | Command Type | Client ID
------|------|--------------|----------
    0 |    5 |      RESERVE | client1
Commit Index: -1
==========================================


===== SERVER 2 LOG (Term 5, Role: Follower) =====
Index | Term | Command Type | Client ID
------|------|--------------|----------
    0 |    5 |      RESERVE | client1
Commit Index: -1
==========================================


===== SERVER 3 LOG (Term 5, Role: Follower) =====
Index | Term | Command Type | Client ID
------|------|--------------|----------
    0 |    5 |      RESERVE | client1
Commit Index: -1
==========================================
```

```
===== SERVER 1 LOG (Term 5, Role: LEADER) =====
Index | Term | Command Type | Client ID
------|------|--------------|----------
    0 |    5 |      RESERVE | client1
Commit Index: 0
==========================================
```

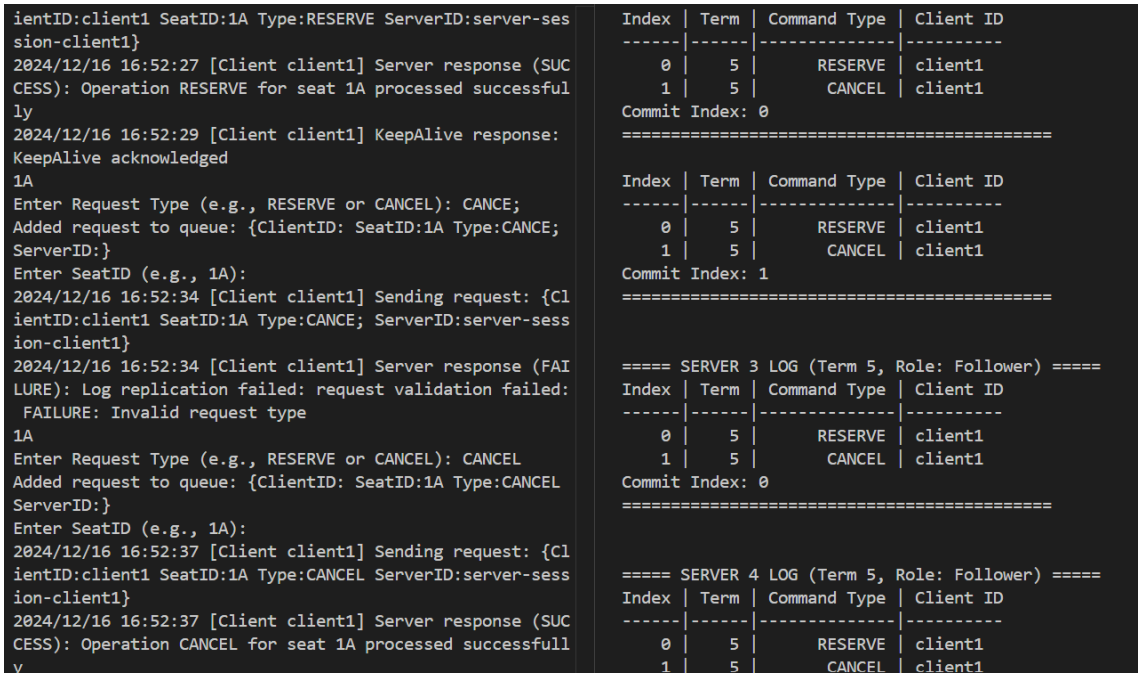
5. The client attempts to reserve an already booked seat (1A) and the server rejects the request as the seat is already occupied:



6. A client attempts to CANCEL a seat booked by a different client:



7. Appending log entries on replica (follower) servers based on valid requests (RESERVE followed by CANCEL):



```
ientID:client1 SeatID:1A Type:RESERVE ServerID:server-ses
sion-client1}
2024/12/16 16:52:27 [Client client1] Server response (SUC
CESS): Operation RESERVE for seat 1A processed successful
ly
2024/12/16 16:52:29 [Client client1] KeepAlive response:
KeepAlive acknowledged
1A
Enter Request Type (e.g., RESERVE or CANCEL): CANCE;
Added request to queue: {ClientID: SeatID:1A Type:CANCE;
ServerID:}
Enter SeatID (e.g., 1A):
2024/12/16 16:52:34 [Client client1] Sending request: {Cl
ientID:client1 SeatID:1A Type:CANCE; ServerID:server-sess
ion-client1}
2024/12/16 16:52:34 [Client client1] Server response (FAI
LURE): Log replication failed: request validation failed:
 FAILURE: Invalid request type
1A
Enter Request Type (e.g., RESERVE or CANCEL): CANCEL
Added request to queue: {ClientID: SeatID:1A Type:CANCEL
ServerID:}
Enter SeatID (e.g., 1A):
2024/12/16 16:52:37 [Client client1] Sending request: {Cl
ientID:client1 SeatID:1A Type:CANCEL ServerID:server-sess
ion-client1}
2024/12/16 16:52:37 [Client client1] Server response (SUC
CESS): Operation CANCEL for seat 1A processed successfull
y
```

```
Index | Term | Command Type | Client ID
------|------|--------------|----------
    0 |    5 |      RESERVE | client1
    1 |    5 |       CANCEL | client1
Commit Index: 0
==========================================

Index | Term | Command Type | Client ID
------|------|--------------|----------
    0 |    5 |      RESERVE | client1
    1 |    5 |       CANCEL | client1
Commit Index: 1
==========================================


===== SERVER 3 LOG (Term 5, Role: Follower) =====
Index | Term | Command Type | Client ID
------|------|--------------|----------
    0 |    5 |      RESERVE | client1
    1 |    5 |       CANCEL | client1
Commit Index: 0
==========================================


===== SERVER 4 LOG (Term 5, Role: Follower) =====
Index | Term | Command Type | Client ID
------|------|--------------|----------
    0 |    5 |      RESERVE | client1
    1 |    5 |       CANCEL | client1
```

Limitations and Future Works

1. The current implementation does not allow clients to book multiple seats concurrently within the same command and requires them to enter separate commands for each action.
2. The implementation needs to be improved to achieve reduced latency when handling a larger number of requests as observed by the significant increase in the average response time when scaling the number of concurrent requests.
3. The Raft Log Replication does not consider all edge cases such as when quorum is not achieved.
4. The load balancer is a single point of failure since we assumed zero faults for the load balancer and handled faults on the servers instead.

References

1. Bai, S., Li, H. (n.d.). Chubby. GitHub. Retrieved October 13, 2024, from https://github.com/sherrybai/chubby/tree/master

2. Yongman. (n.d.). Leto. GitHub. Retrieved October 15, 2024, from https://github.com/yongman/leto

3. Ousterhout, J. (2020). Raft: A consensus algorithm for managing a replicated log. Stanford University. Retrieved December 2, 2024, from https://web.stanford.edu/~ouster/cgi-bin/cs190-winter20/lecture.php?topic=raft

4. Ongaro, D., & Ousterhout, J. (n.d.). Raft: Understandable distributed consensus. Raft. Retrieved December 2, 2024, from https://raft.github.io/

5. The Secret Lives of Data. (n.d.). Raft. Retrieved December 2, 2024, from https://thesecretlivesofdata.com/raft/

6. Burrows, M. (2006). The Chubby lock service for loosely-coupled distributed systems. Google Research. Retrieved October 1, 2024, from https://static.googleusercontent.com/media/research.google.com/en//archive/chubby-osdi06.pdf