UNIVERSITÉ Concordia UNIVERSITY

GINA CODY
SCHOOL OF ENGINEERING
AND COMPUTER SCIENCE

Software Failure Tolerant and Highly Available Distributed Health Care Management System

COMP 6231 – Winter 2024

DISTRIBUTED SYSTEM DESIGN

Concordia University

Department of Computer Science and Software Engineering

Instructor: - R. Jayakumar

Teacher assistance: - Vaibhav Parihar

Group members:

Janmitsinh Panjrolia - 40294468 – janmitsinhpanjrolia@gmail.com

Hardik Gohil – 40294049 – gohilhardik73253@gmail.com

Punamkumar Vekariya – 40290462 – poonamvekariyapv@gmail.com

Abhishek Maurya - 40294111 - abhishek2504maurya@gmail.com

Table of Content

Overview:

This document outlines the design approach for achieving highly available and software failure tolerance in a distributed healthcare management system.

Challenges and Goals:

Distributed systems offer scalability and flexibility but are vulnerable to software failure in individual components. In a critical application like a healthcare management system, such failures can have severe consequences, impacting patient care and potentially compromising sensitive medical records. This design addresses the following goals:

- ♦ Highly available: Ensure the system remains operational and responsive to user requests even during component failures.
- ♦ Software Failure Tolerance: Mitigate the impact of software bugs or crashes on overall system functionality.

Technology Used/ Theories:

Webservice:

- → A web service is essentially a software program that provides functionalities accessible over the internet.
- → Web services act as intermediaries, allowing different applications to talk to each other regardless of their programming languages or operating systems.
- → Web services rely on standardized protocols to ensure smooth communication. These protocols define how data is formatted and transmitted. Common ones include SOAP and XML.
- → Used WSDL that tells you what operations are available, what data format are required.

Replication:

- → Replication in computing involves sharing information to ensure consistency between redundant resources, such as software or hardware components, to improve reliability, fault-tolerance, or accessibility.
- → Replication can be implemented as Passive Replication and Active Replication. Where the requests are sent from the client to Front-end (FE) which acts as the middleware between client and the servers.
- → Each server has replicas which are managed by the Replica Manager (RM). RM is responsible for failure detection and failure recovery.

Total Ordering:

→ Total ordering can be achieved using sequence numbers attached by the sequencer to every request. We use FIFO for the order of the processing. Also, in case of a request lost, any RM receiving new request with any sequence number will check if the new sequence number is after the last executed request, if not; it will ask other RMs for the missing sequence numbers.

UDP Multicast and UDP Unicast:

→ To decrease the number of requests, send through the network, the connection between the sequencer and the RMs, also the connection between RMs is sent through UDP-Multicast.
→ Unicast involves sending data packets from a single source IP address to a single destination IP address.

UDP Reliability:

→ Time out resends: By the FE to overcome the request lost.
→ Multicasting each new request received by an RM to other RMs.
→ If a new request received by an RM was not after the last executed request If a new request received by an RM was not after the last executed request.

Total Ordering using Sequencer:

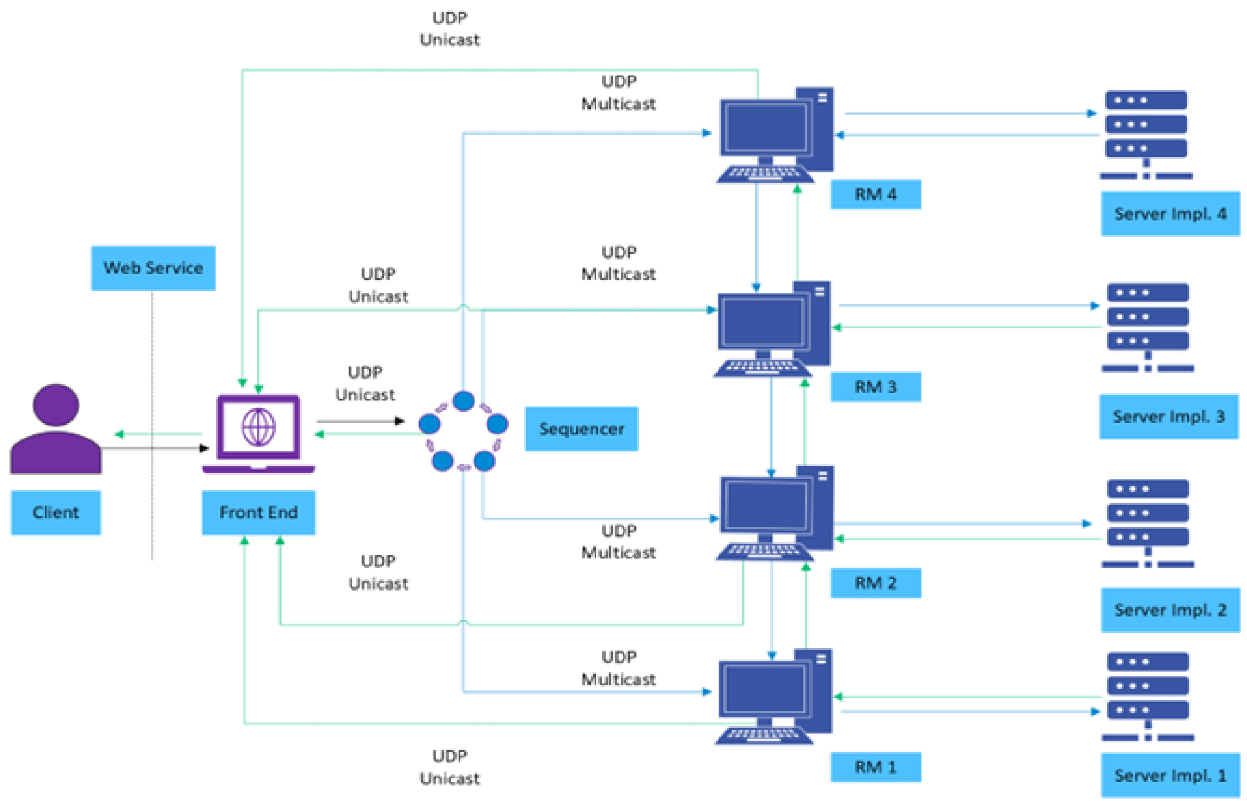
→ To ensure Total Ordering to all the RM the request is transmitted from the FE to the sequencer. The sequencer then generates a unique sequence number and replies to it to the FE which then multicasts to all the RM.
→ Here, client requests are multicast twice. Once, from FE to the RM without the sequence number and other time it generates the sequence number using sequencer for each request and multi-cast to RM again. This cause bandwidth to be overhead on the network as the request is multi-cast twice to the RM.

Kaashoek's Protocol:

→ In Kaashoek's protocol, the FE will send the request from client to the sequencer where sequencer will generate sequences of the client request and multi-cast it.
→ By Kaashoek's Protocol, we can achieve Byzantine fault tolerance in DHMS by leveraging concepts such as voting, view change, commit rules and quorums. It allows DHMS to maintain correctness and consistency even in the presence of Byzantine faults, making it suitable for applications requiring high reliability and fault tolerance.

MVC (Model View Controller) Architecture:

In our architecture, we have Client, Web Service, Front-End (FE), Sequencer and Replica Manager (RM).



Client: This represents the device used to initiate requests to the web service, such as computer.

Front End: This component receives the user's requests and transmits them to the web service.

Sequencer: This part of the system is responsible for managing the order in which requests are processed by the servers. Imagine a queue or waiting line where requests are sorted according to a defined priority.

Server Implementation (RM1, RM2, RM3, RM4): These represent the servers that fulfill the requests received from the front end. They can potentially handle different tasks or specialize in handling specific data. For instance, one server might be responsible for processing login requests, while another might focus on fulfilling booking queries.

Failure Detection and Recovery:

Replica Managers play a critical role in ensuring system reliability and fault tolerance.

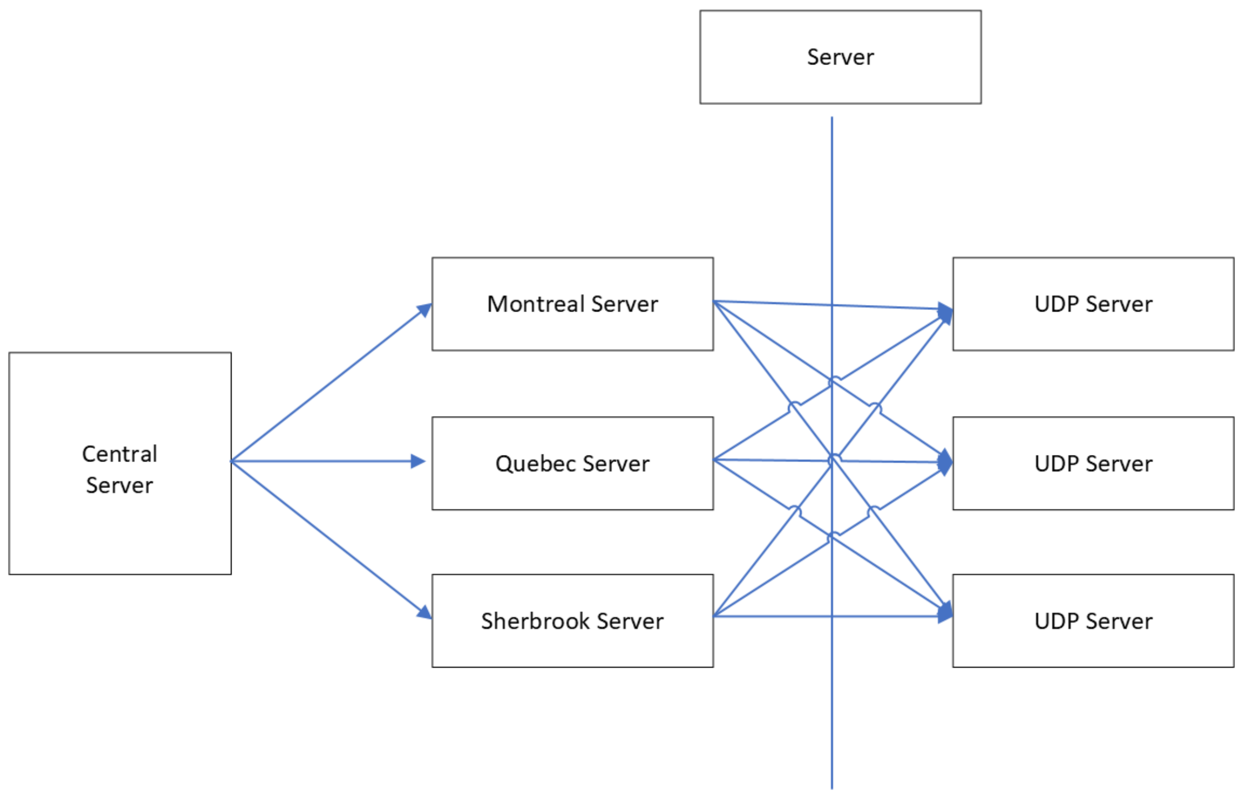
→ *Failure Detection:* RMs employ techniques like heartbeats or timeouts to detect failure within their assigned replicas.

→ *Recovery Actions:* Upon detecting a failure, the RM initiates a recovery protocol for the affected replica. This may involve restarting the replica process or restoring it from a recent checkpoint.

Heartbeat:

→ Heartbeat mechanisms play a crucial role in maintaining the reliability and availability of distributed systems by enabling efficient monitoring and detection of failures.

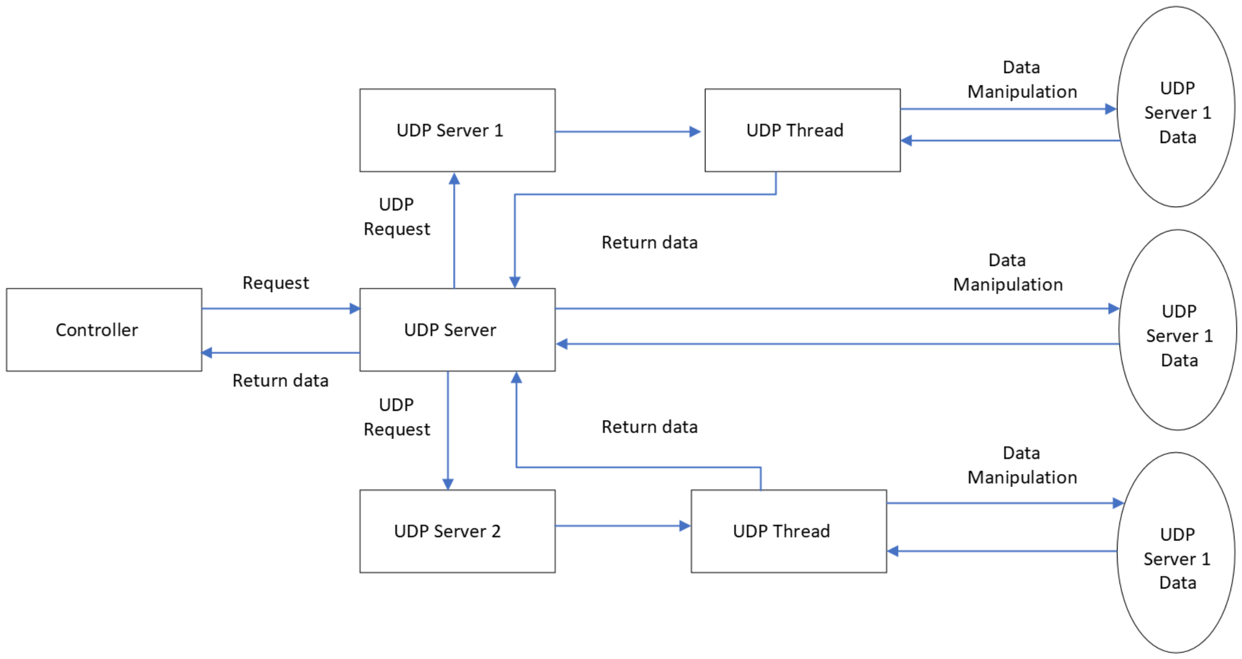
Server Architecture within single Replica:



UDP: This refers to the User Datagram protocol, a communication protocol that transmits data packets over a network without errors and retransmitting lost data packets, UDP prioritizes speed over reliability.

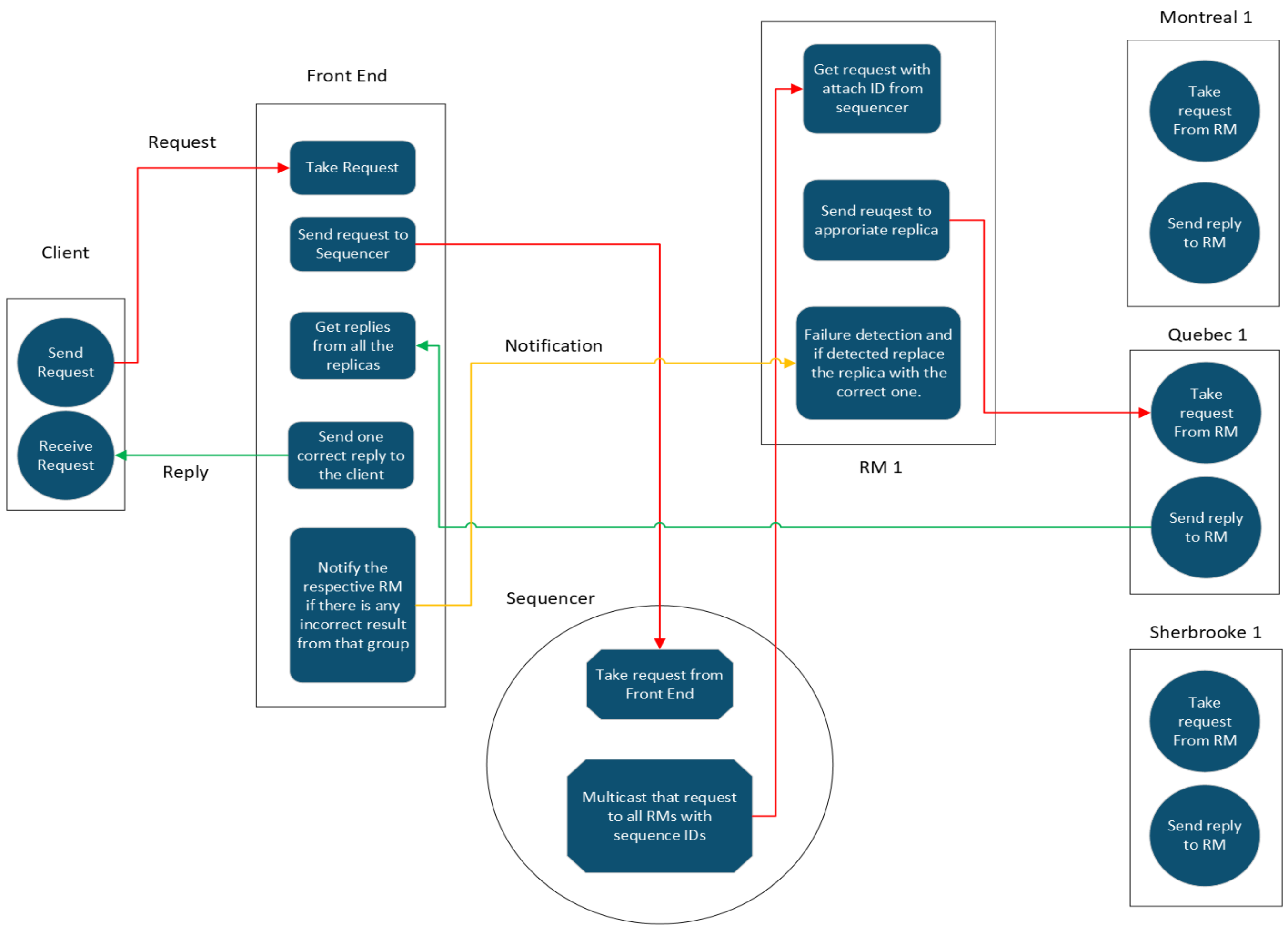
The communication between components appears to be facilitated through UDP Unicast and Multicast protocols. Unicast refers to one-to-one communication, where a single sender transmits data to a single receiver. Multicast, on the other hand, allows for one-to-many communication, where a single sender transmits data to a specific group of receivers.

UDP Server Design:



Dataflow Design:

Dataflow provides a visual representation of the operation of each process and how the data is transmitted to achieve Software Failure Tolerance and Highly Available DHMS.


Client
Send Request
Receive Request
Request
Reply
Front End
Take Request
Send request to Sequencer
Get replies from all the replicas
Send one correct reply to the client
Notify the respective RM if there is any incorrect result from that group
Notification
Sequencer
Take request from Front End
Multicast that request to all RMs with sequence IDs
Get request with attach ID from sequencer
Send reuqest to appropriate replica
Failure detection and if detected replace the replica with the correct one.
RM 1
Montreal 1
Take request From RM
Send reply to RM
Quebec 1
Take request From RM
Send reply to RM
Sherbrooke 1
Take request From RM
Send reply to RM

Client: The client initiates requests for the system.

Front End: The front end receives requests from the client and forwards them to the sequencer.

Sequencer: The sequencer assigns a unique sequence number to each request process request from the sequencer and sends replies to it.

Replicas (RM): Replicas are servers that maintain a copy of the data store. They process requests from the sequencer and send replies to it.

Reply: The reply is the response from a replica to a request.

How to achieve the Highly Available and Fault Tolerance:

Active Replication for Fault Tolerance

The system employs active replication, running four identical replicas of the healthcare management service concurrently. Each replica receives the same user requests and independently processes them, ensuring redundancy and fault tolerance. In the replica failure, the remaining replicas can continue processing requests without service interruption.

Software Fault Tolerance:

To guarantee software fault tolerance, the system implements the following:

- Total Order Execution: A consensus-based total order multicast approach ensures all replicas process requests in the exact order. Each replica returns its result to the front end (FE).
- Result Aggregation: The FE employs a voting mechanism (e.g., majority vote) to aggregate results from the replicas. This ensures only the valid and consistent outcome is delivered to the client.
- Replica Failure Detection: The FE monitors for discrepancies in replica outputs. If a replica consistently produces incorrect results (exceeding a predefined threshold of three times), the FE notifies all Replica Managers (RMs).
- Replica Replacement: Upon notification from the FE, RMs initiate a replacement process for the faulty replica. This may involve restarting the replica or restoring it from a recent checkpoint.

Consensus-based Multicast for High Availability:

A failure-free sequencer serves as the central component for achieving high availability:

- Client Request Submission: The FE forwards client requests to the sequencer.
- Total Order Multicast: The sequencer assigns a unique sequence number to each request and reliably multicasts it, along with FE information, to all RMs.
- Replica Synchronization: This ensures all replicas receive requests in the same order, preventing inconsistencies and data corruption.

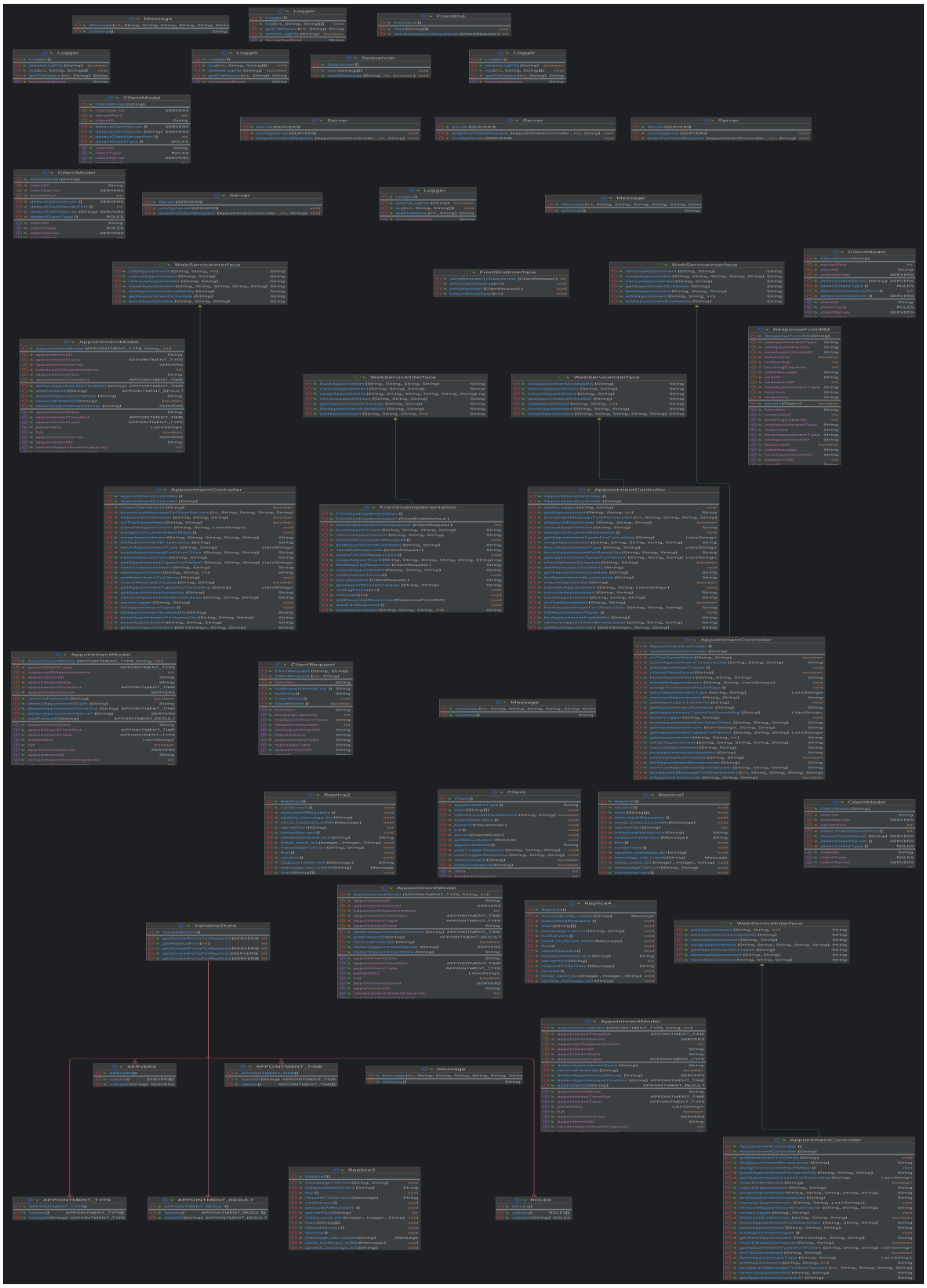
Replica Manager for Failure Detection and Recovery:

Replica Managers (RMs) play a crucial role in maintaining system reliability:

- Failure Detection: RMs utilize techniques like heartbeats or timeouts to detect crashes within their assigned replicas.
- Recovery Actions: Upon detecting a failure, the RM initiates a recovery procedure for the affected replica. This may involve restarting the replica process or restoring it from a recent checkpoint.
- Replica Replacement (Secondary Mechanism): In addition to FE-based detection, RMs can also identify potential replica crashes if a response isn't received within a designated timeframe. RMs can then initiate a consensus protocol to determine if a replica has indeed failed. If most RMs agree on the failure, the faulty replica can be replaced with a healthy one.

By combining these techniques, the system achieves high availability by ensuring service continuity even during replica failures. Additionally, active replication with failure detection and recovery mechanisms mitigates the impact of software errors, promoting overall system fault tolerance.

Class Diagram:



Roles and Responsible

[Student 1] [Janmitsinh Panjrolia – 40294468]

Front End (FE):

In this project, we use a front-end as a median to communicate with the client, RMs and the sequencer. The front-end uses Webservice Interface to send/receive message to/from client. For the system to become fault tolerance or highly available, the FE calculates most responses and detects every/bug due to the response from each RM.

How is it design and integrate?

- **Request formatting:** The Front End (FE) organizes client requests into a standardized format for efficient communication.
- **Sequencer interaction**: The FE sends the formatted request to the sequencer via UDP unicast for tracking and assigning a unique sequence ID.
- **Response waiting**: The FE patiently awaits responses from multiple servers, called Replica Managers (RMs), who handle the request.
- **Dynamic timeout**: The FE employs an adaptable timeout mechanism, starting at 10 seconds and adjusting based on typical response times.
- **Crash detection**: If an RM fails to respond within the timeout multiple times, it's considered crashed, and a notification is sent to all other RMs.
- **Response parsing**: Upon receiving RM responses, the FE carefully analyzes them to identify the majority outcome and detect discrepancies.
- **Bug detection**: If a particular RM consistently disagrees with the majority, a potential software bug is flagged, and again, all RMs are notified.
- **Multicast communication**: The FE uses UDP multicast to efficiently broadcast crash and bug notifications to all RMs, ensuring system-wide awareness and potential corrective actions.

Why choose this approach?

For the Client-FE webservice it seems convenient because by using it we can use different programming languages.

Also, putting all the standard communications and message formats in class which every module can have access to keep the code reworks needed to the minimum since everyone had to implement the same class which strictly forced them to use the standard messages.

[Student 2] [Punam Kumar Vekariya – 40290462]

Replica Manager (RM):

In this project, we used 4 replica managers capable of providing fault tolerance or availability by selecting the feature when the server system is initialized.

How is it design and integrate?

- **Message Sharing**: Upon receiving a message, RMs act as relays, forwarding it to other RMs using multicast for efficient dissemination.
- **Message Storage**: Each RM keeps track of received messages. They are Added to a queue for ordered processing (likely based on priority). Stored in a HashMap using the message's unique sequence ID as the key for easy retrieval.
- **Dedicated Processing Thread**: A separate thread within each RM is responsible for executing messages from the queue, likely prioritizing them based on their importance.
- **RM-Server Interaction:** RMs communicate with three separate servers using a technology called RMI (Remote Method Invocation) to presumably send requests or retrieve data.
- **Standardized Communication**: A dedicated class defines a common format for messages exchanged between RMs and the Front End (FE), ensuring clear and consistent communication across the system.

Message Format:

uniqueSequenceID; FEIPAddress; MessageType; function (bookAppointment,...); userID; newAppointmentID; newAppointmentType; oldAppointmentID; oldAppointmentType; bookingCapacity

Using Message Type in RM we can indicate the type of request (00- Simple message...). Each RM in case there is a bug in any server, will inform which RM has bug. In case the servers of each RM have been crashed, the RM will restart each server and will reload the databases in each server. In case each RM has lost a message, they will ask other RMs to update their HashMap, using the sequenceID, means they ask only the missing sequenceID since they are unique and then replica will start to execute the requests.

Why choose this approach?

By using Webservice for sending requests to server it would be much simpler to compare with UDP. By RM, we can easily manage the fault tolerance and synchronization.

[Student 3] [Abhishek Mauriya – 40294111]

Sequencer:

Sequencer is a middle ware that runs on unicast to get the information from Front-End and uses multi-cast to send the data to 4 RM's.

How is it designed and integrated?

The data sent via a FE is captured and split into parts using semicolon. The first part is Sequencer id along with the rest of the message. This sequencer id is sent back to FE where it verifies the number of attempts made to send the data. On the other hand, the sequencer id is incremented and added to the remaining message along with the IP of FE which is sent to the RM.

[Student 4] [Hardik Gohil – 40294049]

Client program and Test Scenarios:

Client represents the device used to initiate requests to the web service, such as computer. The client initiates requests for the system.

1) **Test for assigning a unique sequence number:**

- Send a request to the sequencer with different parameters and verify that each request is assigned a unique sequence number.
- Send multiple requests to the sequencer at the same time and verify that each request is assigned a unique sequence number.

2) **Test for reliable multicast:**

- Send a request to the sequencer and verify that it is multicast to all replicas.
- Simulate a network failure between the sequencer and one of the replicas, send a request to the sequencer, and verify that the request is still multicast to the other replicas.

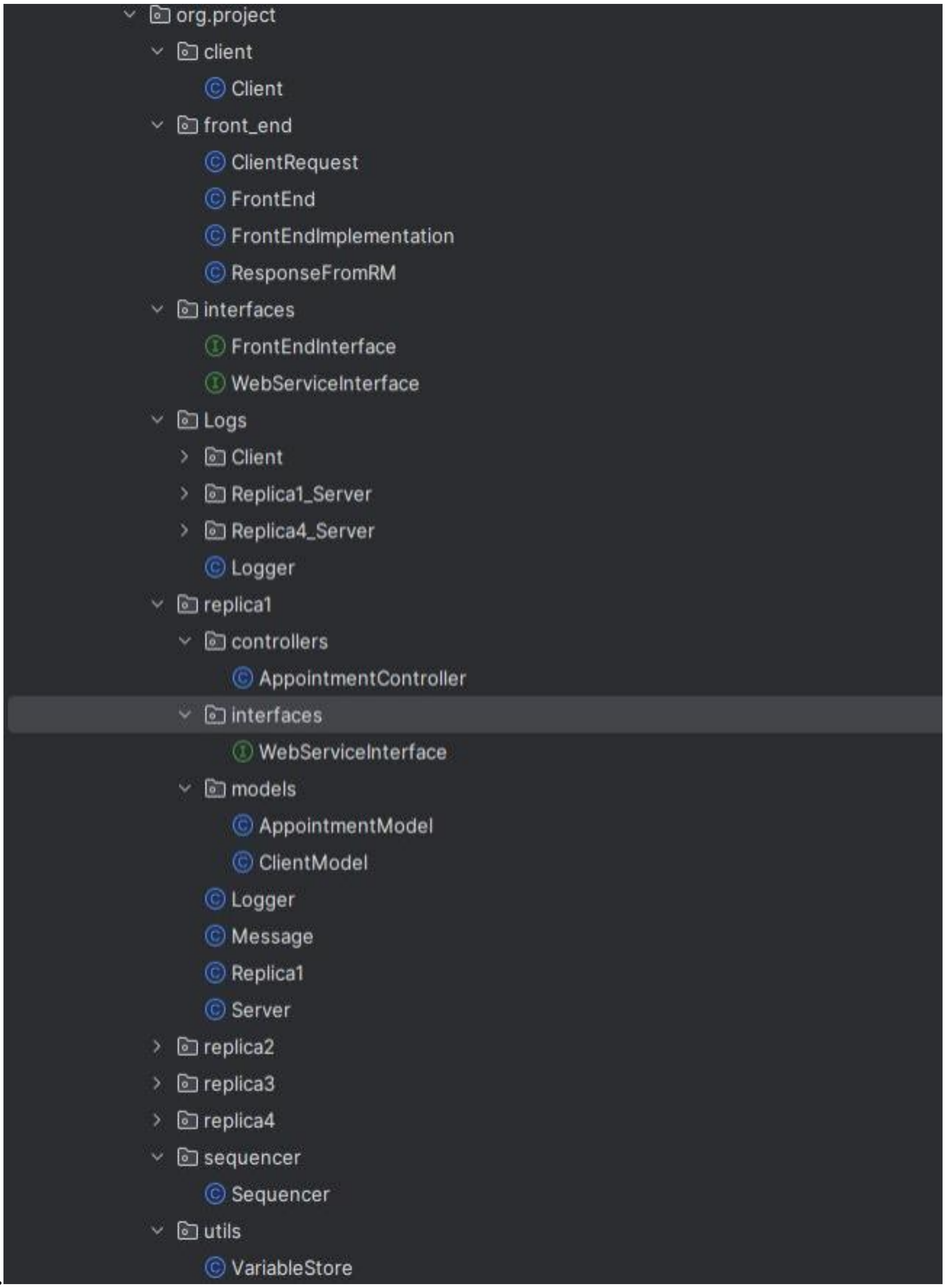
3) **Test for total order execution:**

- Send requests to the sequencer with different parameters and verify that they are executed by the replicas in the same order as they were received by the sequencer.
- Send multiple requests to the sequencer at the same time and verify that they are executed by the replicas in the same order as they were received by the sequencer.

4) **Test all the possible admin and patient actions:**

- Test Admin's add appointments, list appointments Availability, get appointments schedule, cancel appointments, book appointments and swap appointments for patient.

Folder Structure

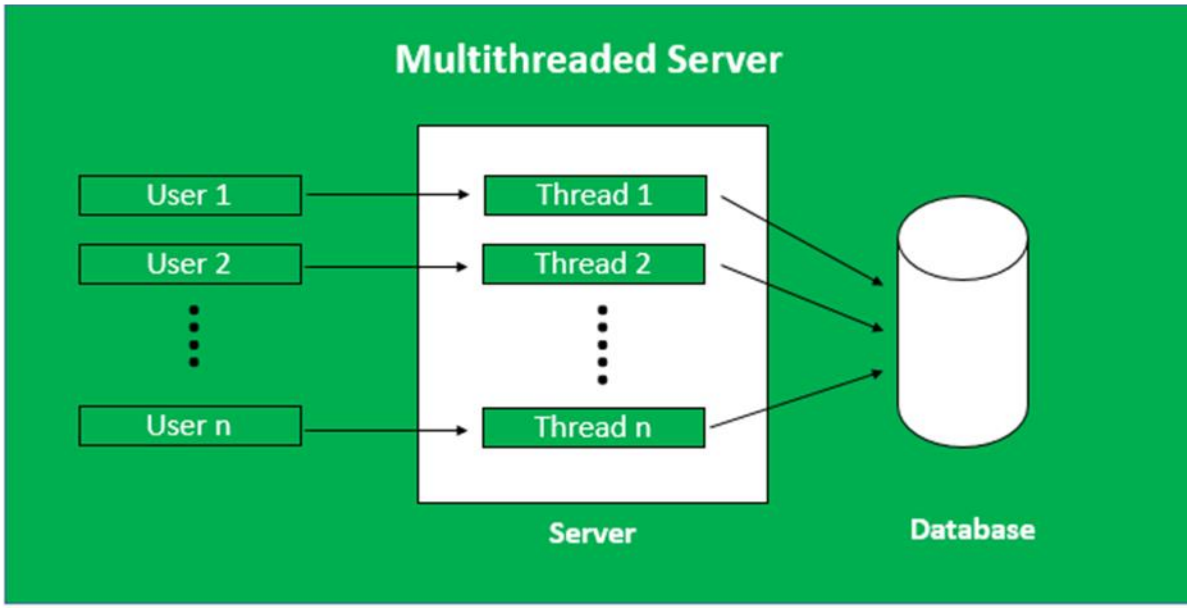


```
org.project
  client
    Client
  front_end
    ClientRequest
    FrontEnd
    FrontEndImplementation
    ResponseFromRM
  interfaces
    FrontEndInterface
    WebServiceInterface
  Logs
    Client
    Replica1_Server
    Replica4_Server
    Logger
  replica1
    controllers
      AppointmentController
    interfaces
      WebServiceInterface
    models
      AppointmentModel
      ClientModel
    Logger
    Message
    Replica1
    Server
  replica2
  replica3
  replica4
  sequencer
    Sequencer
  utils
    VariableStore
```

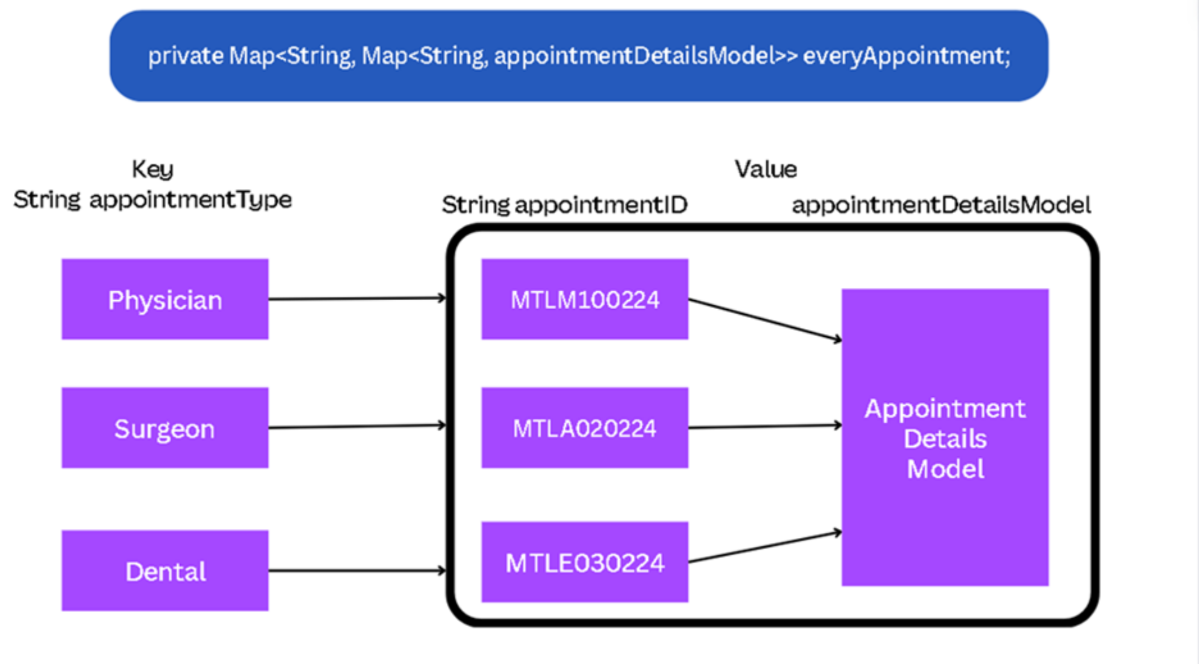
.

Data Structure:

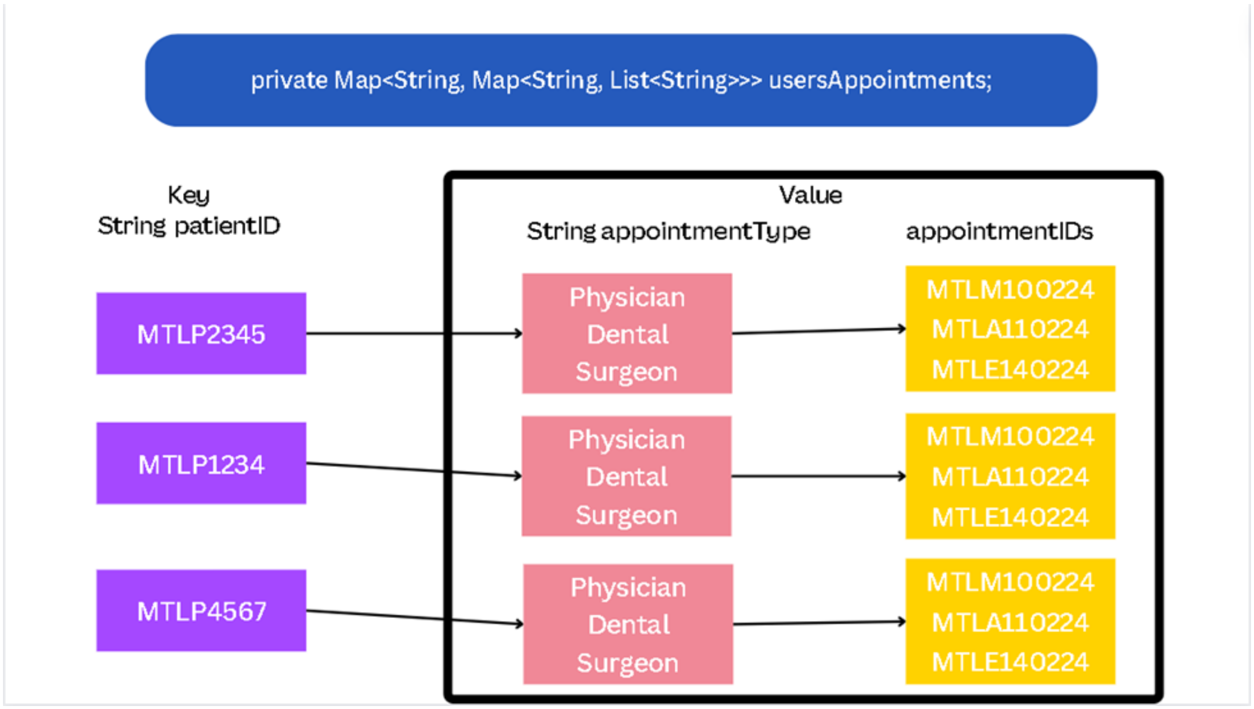
1) key: serverName value: serverPort



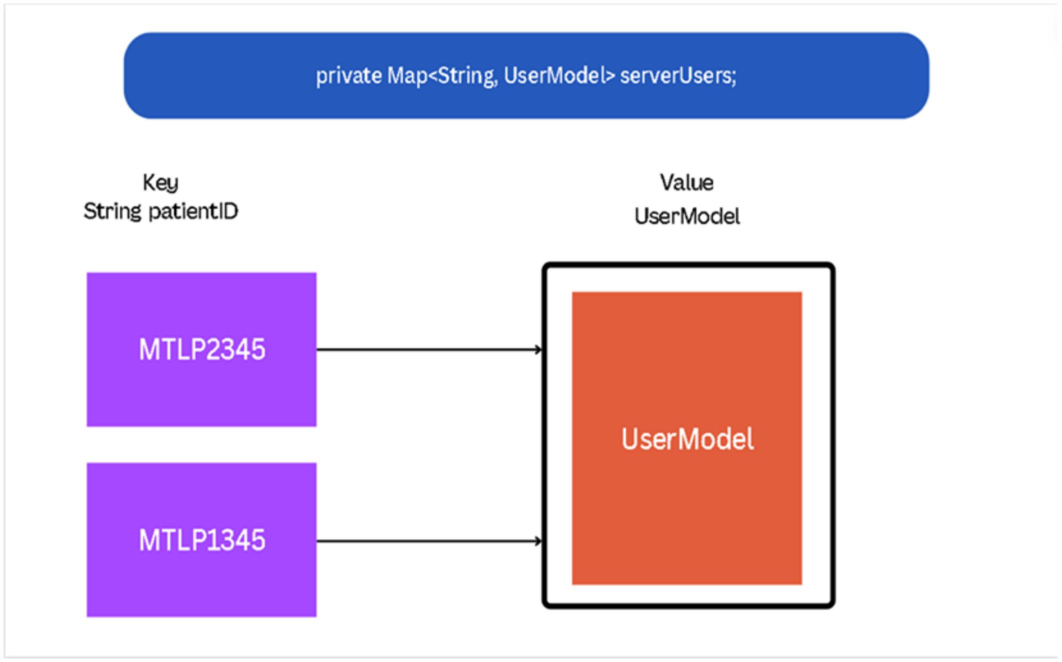
2) Key: AppointmentType Value: <appointmentID, appointmentDetailsModel>


```
private Map<String, Map<String, appointmentDetailsModel>> everyAppointment;
```

3) Key: patientID value: <appointmentType, appointmentIDs>



4) Key: patientID value: <userModel>



Algorithm

Front End:

1. **Define message format:**
   - String messageFormat = "Sequenceid;FEIpAddress;MessageType;function(addAppointment,...);userID;newAppointmentD;newAppointmentType;oldAppointmentID; oldAppointmentType; bookingCapacity"

2. **Define timeout parameters:**
```
- int timeout = 10000; // 10 seconds
- int numOfRetries = 3;
```

3. **Define multicast group for notifying RMs:**
```
- String multicastGroup = "230.1.1.10";
```

4. **Serialize request from client and send to sequencer:**
```
- String serializedRequest = serializeRequest(clientRequest);
- String sequencerResponse = sendToSequencer(serializedRequest);
```

5. **Parse sequence ID from sequencer response:**
```
- int sequenceId = parseSequenceId(sequencerResponse);
```

6. **Wait for responses from RMs:**
```
- List<String> rmResponses = new ArrayList<>();
- for each RM in RMs:
        - String response = null;
        - int t numOfAttempts = 0;
        - while (response == null && numOfAttempts < numOfRetries):
                - response = sendToRM(rm, serializedRequest, timeout);
                - numOfAttempts ++;
        - rmResponses.add(response);
```

7. **Parse RM response and detect bugs/crashes:**
```
- List<RmResponse> parsedResponses = new ArrayList<>();
- for each rmResponse in rmResponses:
```

```
            - parsedResponses.add(parseResponse(rmResponse));
            - int numOfCrashes = 0;
            - int num Of Bugs = 0;
            - RmResponse majorityResponse = getMajorityResponse(parsedResponses);
            - for each parsedResponse in parsedResponses:
                    - if parsedResponse not equals majorityResponse:
                            - if parsedResponse.isCrashed():
                              - num Of Crashes++;
                              - notifyRM(multicastGroup, parsedResponse.getRmId(), "23");
                    - else:
                            - num Of Bugs++;
                            - notifyRM(multicastGroup, parsedResponse.getRmId(), "13");
```

8. **Send response to client**:

```
   - CommonOutput output = createOutput(numOfCrashes, numOfBugs,
majorityResponse);
   - String serializedOutput = serializeOutput(output);
   - sendToClient(clientIpAddress, serializedOutput);
```

Replica Manager:

1. **Initialize the RM:**

```
  - sequence_id = 0
  - queue = create_priority_queue()
  - hash_map = create_hash_map()
```

2. **Loop to receive and process messages:**

```
  - while true:
    - message = receive_message_from_sequencer()
    - multicast_message_to_RMs(message)
    - add_message_to_queue_and_hashmap(message, queue, hash_map)
```

3. **Separate thread to execute messages from the queue:**

```
  - function execute_messages():
    - while true:
      - message = get_highest_priority_message(queue)
      - execute_message_on_servers(message)
      - remove_message_from_hashmap(message, hash_map)
```

4. **Function to handle lost messages:**

```
  - function handle_lost_message(sequence_id):
    - multicast_request_for_missing_sequence_id(sequence_id)
    - responses = wait_for_responses_from_other_RMs()
```

- update_local_hashmap_with_missing_messages(responses)

Test Scenarios: (We will add testing data and results later, as this document is written before the implementation)

Test 1: Concurrency Test Scenarios:

→ We will add 3/4 threads that run on frontend and perform FIFO sequencing by the Sequencer. All the invocations take place according to order of received requests. Every time output will be different based on the order of received requests. Requests are having sequence id and based on that conclusion of successful synchronized concurrent requests invocation are stated.

Test 2: Software bug test case/ non-malicious Byzantine:

→ The Sequencer can replace the replica if it gets three times successively incorrect response with the new replica. The new replica still preserves the state of the old replica. This will be achieved using a data restoring mechanism in which data from any other replica and correct the data of the respective server subsystem.

→ In the Distributed Health Care Management system, this software bug will perform as three successively incorrect responses corresponding with requests id 5,6 and 7. After, the 7th request data restore mechanism automatically takes place between replica 1 or replica 2 to the replica 3. This error is implemented into replica 3.

→ Also, meanwhile servers are getting restored, requests may come but will not proceed till restoration is completed.

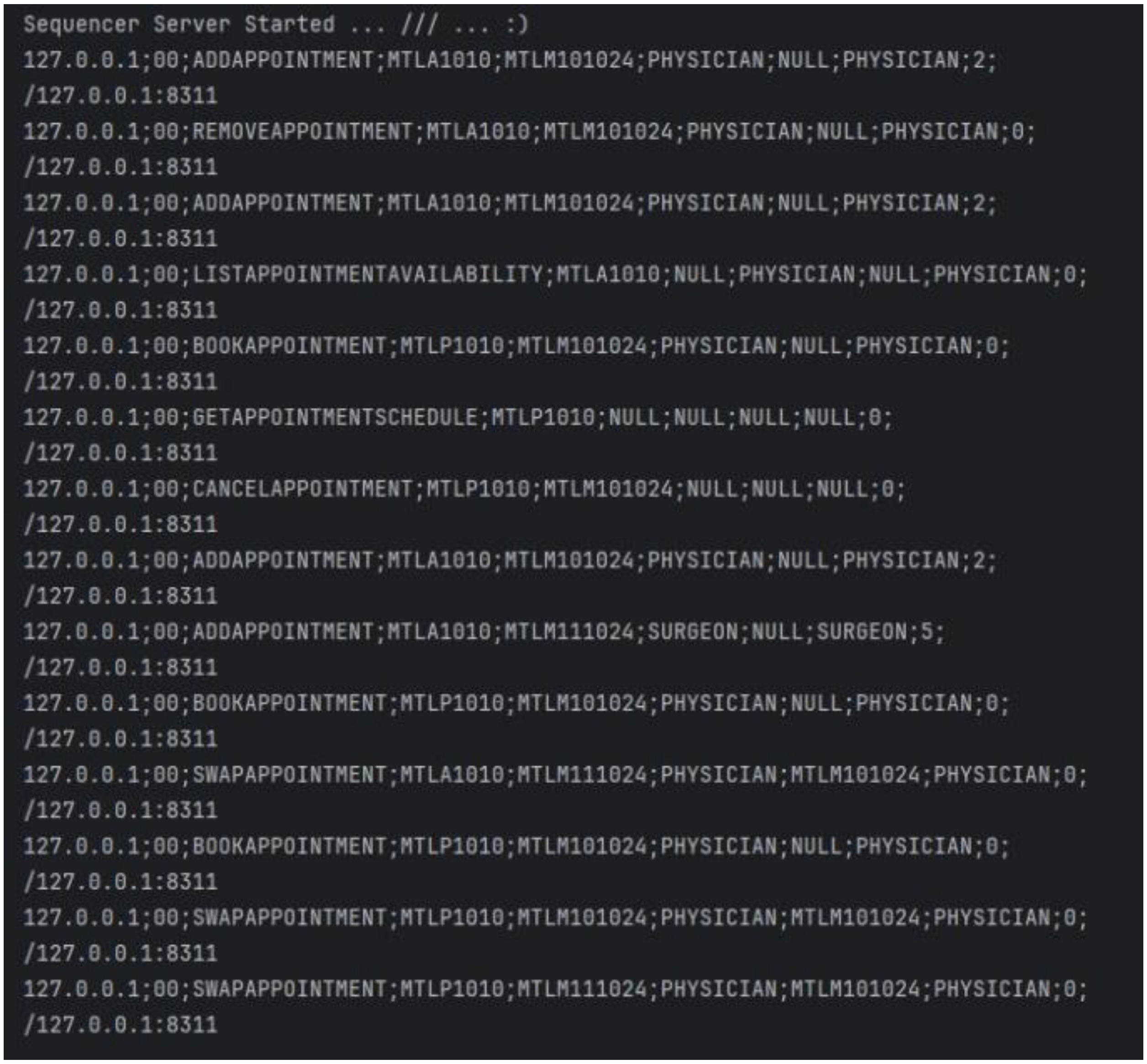
Test 3: Sequencer:

→ Simulate network failures to ensure that the sequencer can recover and resume sequencing without data loss.

→ Test the recovery mechanism of the sequencing to ensure that it can reconcile any missed sequences after a failure.

→ Evaluate the scalability of the sequencer by increasing the number of nodes sending requests simultaneously.

→ Verify that the sequencer maintains accurate sequencing records and does not skip or duplicate sequences.



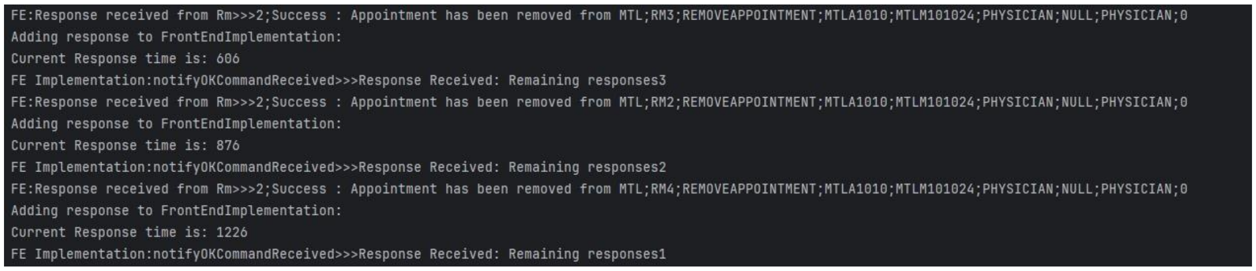
```
Sequencer Server Started ... /// ... :)
127.0.0.1;00;ADDAPPOINTMENT;MTLA1010;MTLM101024;PHYSICIAN;NULL;PHYSICIAN;2;
/127.0.0.1:8311
127.0.0.1;00;REMOVEAPPOINTMENT;MTLA1010;MTLM101024;PHYSICIAN;NULL;PHYSICIAN;0;
/127.0.0.1:8311
127.0.0.1;00;ADDAPPOINTMENT;MTLA1010;MTLM101024;PHYSICIAN;NULL;PHYSICIAN;2;
/127.0.0.1:8311
127.0.0.1;00;LISTAPPOINTMENTAVAILABILITY;MTLA1010;NULL;PHYSICIAN;NULL;PHYSICIAN;0;
/127.0.0.1:8311
127.0.0.1;00;BOOKAPPOINTMENT;MTLP1010;MTLM101024;PHYSICIAN;NULL;PHYSICIAN;0;
/127.0.0.1:8311
127.0.0.1;00;GETAPPOINTMENTSCHEDULE;MTLP1010;NULL;NULL;NULL;NULL;0;
/127.0.0.1:8311
127.0.0.1;00;CANCELAPPOINTMENT;MTLP1010;MTLM101024;NULL;NULL;NULL;0;
/127.0.0.1:8311
127.0.0.1;00;ADDAPPOINTMENT;MTLA1010;MTLM101024;PHYSICIAN;NULL;PHYSICIAN;2;
/127.0.0.1:8311
127.0.0.1;00;ADDAPPOINTMENT;MTLA1010;MTLM111024;SURGEON;NULL;SURGEON;5;
/127.0.0.1:8311
127.0.0.1;00;BOOKAPPOINTMENT;MTLP1010;MTLM101024;PHYSICIAN;NULL;PHYSICIAN;0;
/127.0.0.1:8311
127.0.0.1;00;SWAPAPPOINTMENT;MTLA1010;MTLM111024;PHYSICIAN;MTLM101024;PHYSICIAN;0;
/127.0.0.1:8311
127.0.0.1;00;BOOKAPPOINTMENT;MTLP1010;MTLM101024;PHYSICIAN;NULL;PHYSICIAN;0;
/127.0.0.1:8311
127.0.0.1;00;SWAPAPPOINTMENT;MTLP1010;MTLM101024;PHYSICIAN;MTLM101024;PHYSICIAN;0;
/127.0.0.1:8311
127.0.0.1;00;SWAPAPPOINTMENT;MTLP1010;MTLM111024;PHYSICIAN;MTLM101024;PHYSICIAN;0;
/127.0.0.1:8311
```

Test 4: Front End:

→ Test for session timeout and reauthorization.

→ Test responsiveness of the front end under various load conditions.

→ If a replica consistently produces incorrect results (exceeding a predefined threshold of three times), the FE notifies all Replica Managers (RMs).



```
FE:Response received from Rm>>>2;Success : Appointment has been removed from MTL;RM3;REMOVEAPPOINTMENT;MTLA1010;MTLM101024;PHYSICIAN;NULL;PHYSICIAN;0
Adding response to FrontEndImplementation:
Current Response time is: 606
FE Implementation:notifyOKCommandReceived>>>Response Received: Remaining responses3
FE:Response received from Rm>>>2;Success : Appointment has been removed from MTL;RM2;REMOVEAPPOINTMENT;MTLA1010;MTLM101024;PHYSICIAN;NULL;PHYSICIAN;0
Adding response to FrontEndImplementation:
Current Response time is: 876
FE Implementation:notifyOKCommandReceived>>>Response Received: Remaining responses2
FE:Response received from Rm>>>2;Success : Appointment has been removed from MTL;RM4;REMOVEAPPOINTMENT;MTLA1010;MTLM101024;PHYSICIAN;NULL;PHYSICIAN;0
Adding response to FrontEndImplementation:
Current Response time is: 1226
FE Implementation:notifyOKCommandReceived>>>Response Received: Remaining responses1
```

Test 5: Login

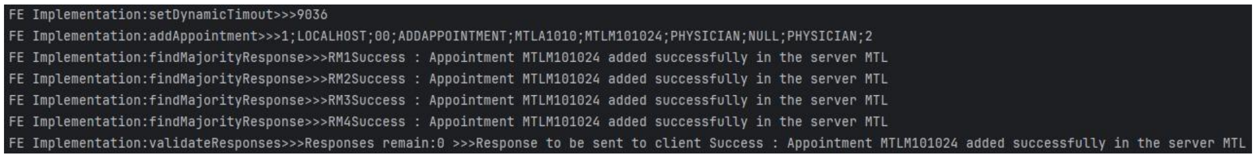
- Verify that a user has a valid Appointment AdminID and PatientID.
- Successful login with valid Admin ID.
- Successful login with valid Patient ID.

Test 2: Logout

- The user should be able to Log0ut when entering the number of the Logout button from a menu item

Test 3: addAppointment [by Admin]

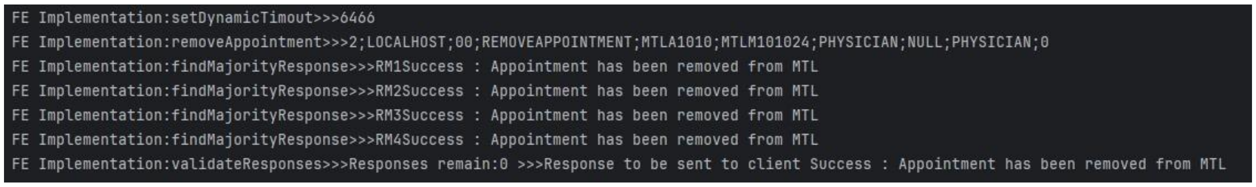
- Test with Invalid AppointmentID
- Add a New AppointmentID
- Add an existing appointment with a lower capacity
- Add an existing appointment with a higher capacity
- Add duplicate appointments to test duplication
- Add an appointment from another server to check the error.
- Add an appointment to the current server.



```
FE Implementation:setDynamicTimout>>>9036
FE Implementation:addAppointment>>>1;LOCALHOST;00;ADDAPPOINTMENT;MTLA1010;MTLM101024;PHYSICIAN;NULL;PHYSICIAN;2
FE Implementation:findMajorityResponse>>>RM1Success : Appointment MTLM101024 added successfully in the server MTL
FE Implementation:findMajorityResponse>>>RM2Success : Appointment MTLM101024 added successfully in the server MTL
FE Implementation:findMajorityResponse>>>RM3Success : Appointment MTLM101024 added successfully in the server MTL
FE Implementation:findMajorityResponse>>>RM4Success : Appointment MTLM101024 added successfully in the server MTL
FE Implementation:validateResponses>>>Responses remain:0 >>>Response to be sent to client Success : Appointment MTLM101024 added successfully in the server MTL
```

Test 4: removeAppointment [by Admin]

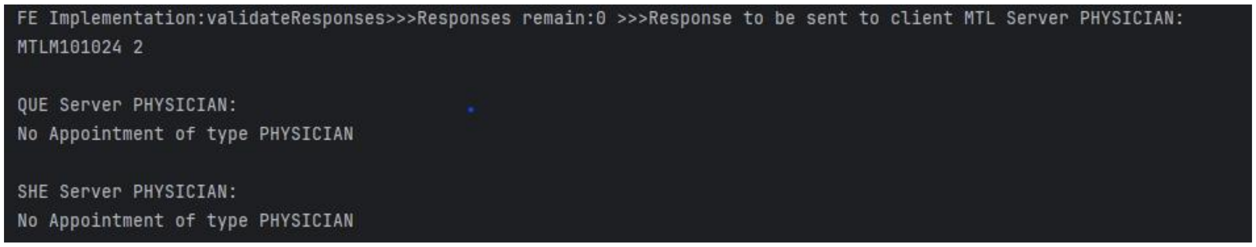
- Attempt to remove an Appointment with an invalid ID.
- Trying to remove a non-existent Appointment.
- Removing an Appointment without any registered participants.
- Removing an Appointment with registered participants and handling reallocation.
- Add Appointments from other servers.



```
FE Implementation:setDynamicTimout>>>6466
FE Implementation:removeAppointment>>>2;LOCALHOST;00;REMOVEAPPOINTMENT;MTLA1010;MTLM101024;PHYSICIAN;NULL;PHYSICIAN;0
FE Implementation:findMajorityResponse>>>RM1Success : Appointment has been removed from MTL
FE Implementation:findMajorityResponse>>>RM2Success : Appointment has been removed from MTL
FE Implementation:findMajorityResponse>>>RM3Success : Appointment has been removed from MTL
FE Implementation:findMajorityResponse>>>RM4Success : Appointment has been removed from MTL
FE Implementation:validateResponses>>>Responses remain:0 >>>Response to be sent to client Success : Appointment has been removed from MTL
```

Test 5: listAppointmentAvailability [by Admin + Patient]

- Give a list of available appointment



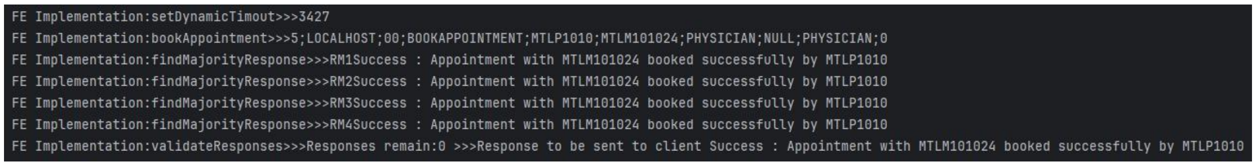
```
FE Implementation:validateResponses>>>Responses remain:0 >>>Response to be sent to client MTL Server PHYSICIAN:
MTLM101024 2

QUE Server PHYSICIAN:
No Appointment of type PHYSICIAN

SHE Server PHYSICIAN:
No Appointment of type PHYSICIAN
```

Test 6: bookAppointment [by Admin + Patient]

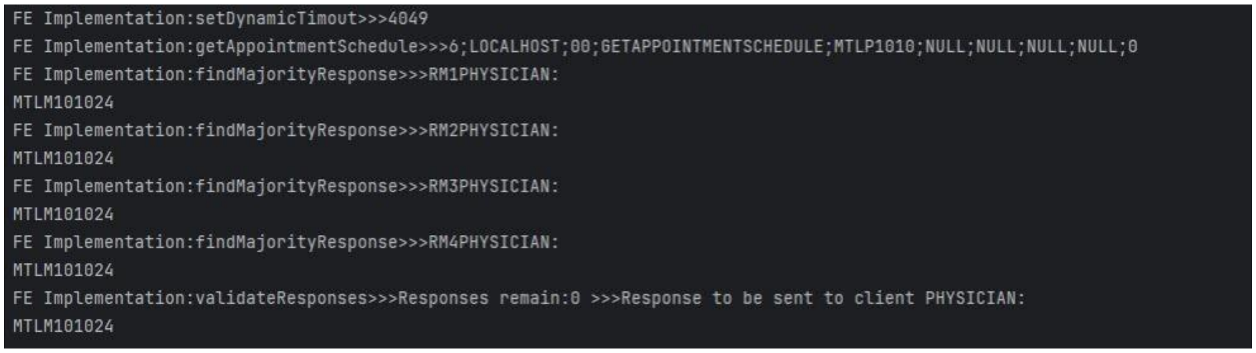
- Book an appointment on the current server
- Book appointments on other servers with a weekly limit of three.
- Book the same appointmentID which has the same appointment type.
- Book with an invalid AppointmentID. (not able to book)
- Book the same appointmentID but a different appointmentType. (not able to book)



```
FE Implementation:setDynamicTimout>>>3427
FE Implementation:bookAppointment>>>5;LOCALHOST;00;BOOKAPPOINTMENT;MTLP1010;MTLM101024;PHYSICIAN;NULL;PHYSICIAN;0
FE Implementation:findMajorityResponse>>>RM1Success : Appointment with MTLM101024 booked successfully by MTLP1010
FE Implementation:findMajorityResponse>>>RM2Success : Appointment with MTLM101024 booked successfully by MTLP1010
FE Implementation:findMajorityResponse>>>RM3Success : Appointment with MTLM101024 booked successfully by MTLP1010
FE Implementation:findMajorityResponse>>>RM4Success : Appointment with MTLM101024 booked successfully by MTLP1010
FE Implementation:validateResponses>>>Responses remain:0 >>>Response to be sent to client Success : Appointment with MTLM101024 booked successfully by MTLP1010
```

Test 7: getAppointmentSchedule [by Admin + Patient]

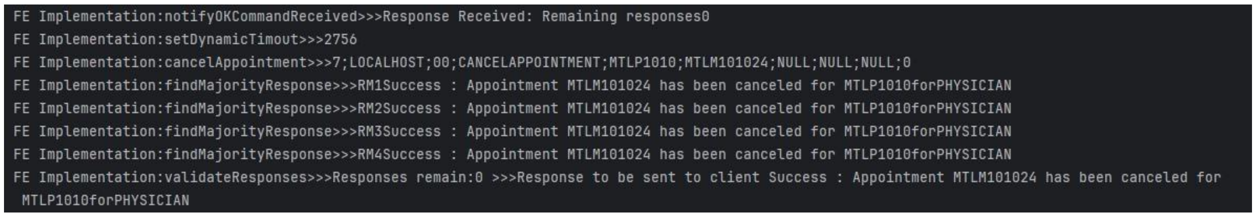
- 1. Get the booking schedule
- 2. Try with an invalid patientID.
- 3. Should give all appointments of patients in the current server.



```
FE Implementation:setDynamicTimout>>>4049
FE Implementation:getAppointmentSchedule>>>6;LOCALHOST;00;GETAPPOINTMENTSCHEDULE;MTLP1010;NULL;NULL;NULL;NULL;0
FE Implementation:findMajorityResponse>>>RM1PHYSICIAN:
MTLM101024
FE Implementation:findMajorityResponse>>>RM2PHYSICIAN:
MTLM101024
FE Implementation:findMajorityResponse>>>RM3PHYSICIAN:
MTLM101024
FE Implementation:findMajorityResponse>>>RM4PHYSICIAN:
MTLM101024
FE Implementation:validateResponses>>>Responses remain:0 >>>Response to be sent to client PHYSICIAN:
MTLM101024
```

Test 8: cancelAppointment [by Admin + Patient]

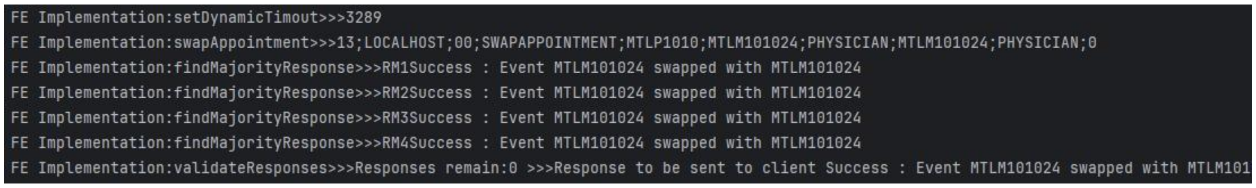
- Cancel a booked Appointment on the current server.
- Cancel a booked Appointment on other servers.
- Canceling a non-registered Appointment.
- Try cancellation of Appointments with an invalid AppointmentID.



```
FE Implementation:notifyOKCommandReceived>>>Response Received: Remaining responses0
FE Implementation:setDynamicTimout>>>2756
FE Implementation:cancelAppointment>>>7;LOCALHOST;00;CANCELAPPOINTMENT;MTLP1010;MTLM101024;NULL;NULL;NULL;0
FE Implementation:findMajorityResponse>>>RM1Success : Appointment MTLM101024 has been canceled for MTLP1010forPHYSICIAN
FE Implementation:findMajorityResponse>>>RM2Success : Appointment MTLM101024 has been canceled for MTLP1010forPHYSICIAN
FE Implementation:findMajorityResponse>>>RM3Success : Appointment MTLM101024 has been canceled for MTLP1010forPHYSICIAN
FE Implementation:findMajorityResponse>>>RM4Success : Appointment MTLM101024 has been canceled for MTLP1010forPHYSICIAN
FE Implementation:validateResponses>>>Responses remain:0 >>>Response to be sent to client Success : Appointment MTLM101024 has been canceled for
 MTLP1010forPHYSICIAN
```

Test 9: swapAppointment [by Admin + Patient]

- Swap Appointments on the current server
- Swap Appointment on a different server
- Swap Appointments of diverse types.
- Swap Appointments on the same appointmentID.



```
FE Implementation:setDynamicTimout>>>3289
FE Implementation:swapAppointment>>>13;LOCALHOST;00;SWAPAPPOINTMENT;MTLP1010;MTLM101024;PHYSICIAN;MTLM101024;PHYSICIAN;0
FE Implementation:findMajorityResponse>>>RM1Success : Event MTLM101024 swapped with MTLM101024
FE Implementation:findMajorityResponse>>>RM2Success : Event MTLM101024 swapped with MTLM101024
FE Implementation:findMajorityResponse>>>RM3Success : Event MTLM101024 swapped with MTLM101024
FE Implementation:findMajorityResponse>>>RM4Success : Event MTLM101024 swapped with MTLM101024
FE Implementation:validateResponses>>>Responses remain:0 >>>Response to be sent to client Success : Event MTLM101024 swapped with MTLM101
```