Distributed Movie Ticket Booking System

Design Document

Project Group - 7

Darshil Patil - 40233875

Rovian Dsouza - 40230102

Vishwas Tomar - 40254026

Omer Sayem - 40226505

Design Explanation

There will be a total of four replicas, and each of these replicas will be equipped with four replica managers to manage requests.

The client will initiate a request to the front end, which will in turn send the request to the sequencer. The sequencer will multicast the request to all of the replica managers and will assign a unique sequence ID to each request. The replica managers will then initialize the replicas, and they will also be responsible for detecting any failures and attempting to recover from them.

After the replicas are initialized, they will start receiving requests from the client. The results of these requests will be forwarded back to the front end, which will select the majority of the results. If any server produces an incorrect output more than three times, it should be rebooted. Similarly, if any server takes longer than the designated timeout period to respond, it should also be rebooted.

The system satisfies two important criteria, namely fault tolerance and replica recovery.

Replica recovery - The replica managers play a crucial role in detecting any failures that may occur within the replicas. In the event of a failure, the replica manager responsible for that replica will initiate a recovery procedure to restore the replica to its previous state. This ensures that the system remains operational even if one or more replicas experience a failure, thereby minimizing the impact on the overall performance of the system.

How will we achieve failure tolerance and High availability?

To achieve both Software Failure Tolerance and High Availability in our distributed movie ticket booking system, we can use ***active replication with a consensus-based approach for total order multicast.***

Active replication involves running multiple replicas of the same service in parallel, each of which receives the same inputs and produces the same outputs. In the event of a failure of one replica, the others can continue to provide the service without interruption.

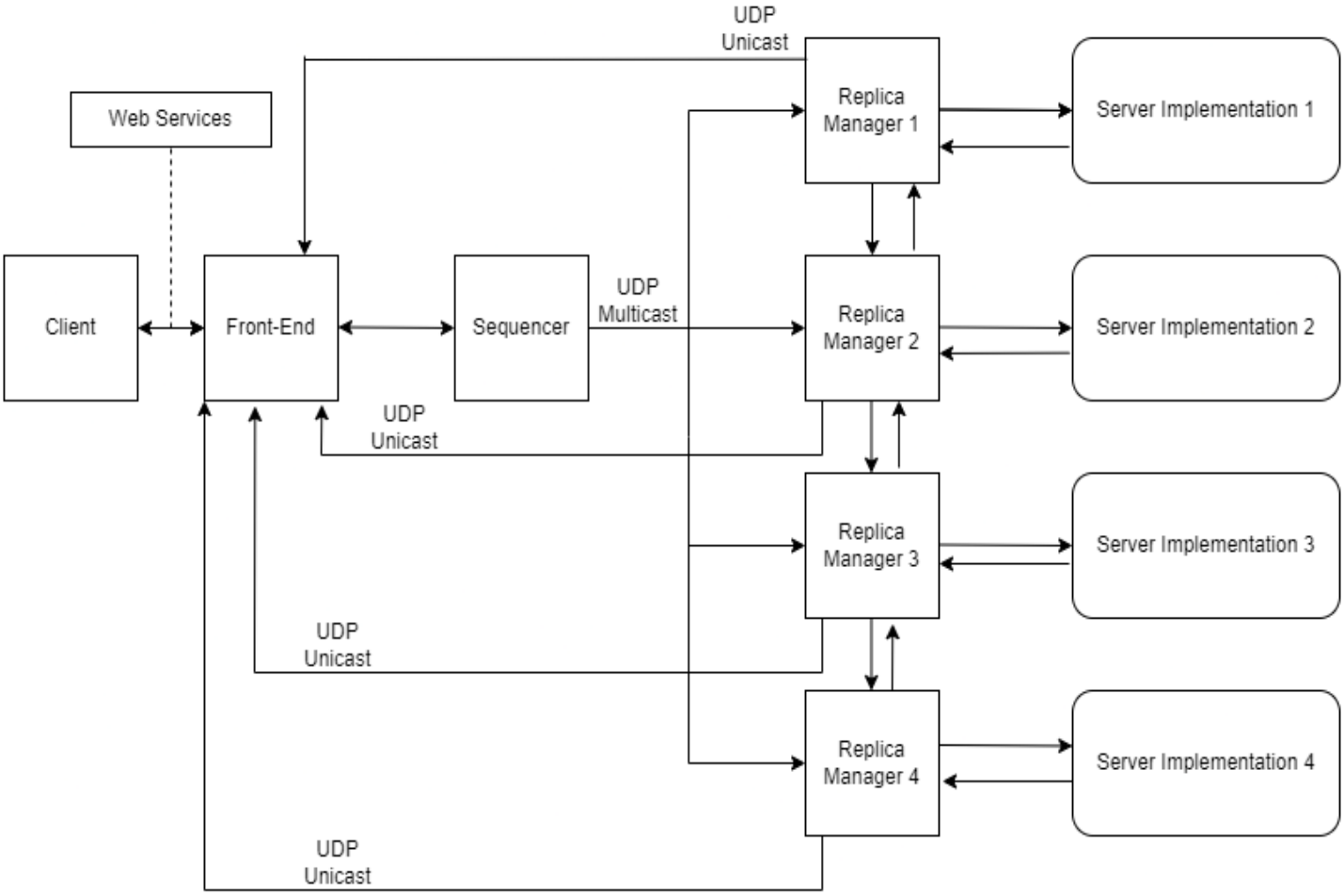
For Software Failure Tolerance, we can ensure that all replicas execute client requests in total order, and return the results back to the front end (FE), which returns a single correct result back to the client as soon as two identical (correct) results are received from the replicas. If any one of the replicas produces an incorrect result, the FE informs all the Replica Managers (RMs) about that replica. If the same replica produces incorrect results for three consecutive client requests, then the RMs replace that replica with another correct one.

For High Availability, we can use a consensus-based approach for total order multicast. The front end (FE) sends client requests to a failure-free sequencer, which assigns a unique sequence number to the request and reliably multicasts the request with the sequence number and FE information to all the server replicas. The sequencer ensures that all replicas receive requests in the same order and that no request is missed or duplicated.

In addition, we can use the Replica Managers (RMs) to detect and recover from a single process crash. If the FE does not receive the result from a replica within a reasonable amount of time, the RM can suspect that the replica may have crashed and inform all the RMs of the potential crash. The RMs can then check the replica that did not produce the result and replace it with another working replica if they agree among themselves that the replica has crashed.

By combining active replication, consensus-based total order multicast, and Replica Managers for failure detection and recovery, we can achieve both Software Failure Tolerance and High Availability in our distributed movie ticket booking system.

Design



Members Task

Darshil Patil – Front End

Omer Sayem – Replica Manager

Rovian Dsouza – Sequencer

Vishwas Tomar – Test Cases for Failure Free Sequencer

Detailed Report

1) Front End (Darshil Patil)

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

What was done?

1. Basically, the FE serializes the request from client in a message format which was standardized by the
2. It is sent to the sequencer and then to the RM.
3. To create dynamic timeout for the FE to wait for the response from RMs, depending on the longest wait time.

2) Replica Manager (Omer Sayem)

We will have replica managers in this project. These RM's will have the below task:

- Create and initialize the actively replicated server subsystem.
- Implement failure detection and recovery for both types of failures (single crash and single software).
- Detects if any replica produces incorrect results or fails to respond in the designated timeout period.
- Initiates recovery procedures to restore the replica to its previous state if a failure is detected.

To achieve high availability, a mechanism is put in place to detect server crashes, which is done by the front-end component. If no response is received from a replica manager (RM) for three consecutive times, or if no heartbeat is received from the RM for a certain period of time (e.g., 10 seconds), then the front-end component initiates a replica recovery process for the affected RM. The replica recovery mode is a state in which the RM is being recovered. During this state, the RM will ask other RMs for all the processed requests. After the recovery process is complete and all past requests have been executed, the RM is considered up and running again.

The connection between RMs and Sequencer is using the multicast receiver between each RM and FE with UDP unicast. Between each RM also the communication is happening using Multicast receiver.

3) Sequencer (Rovian Dsouza)

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 3 RM's.
Sequencer is equipped with a datagram Socket and INET Address from FE.
The received message is split and put in to usable parts then one part of the message is sent back to FE with the same address from which Sequencer received the data and other part along with IP and incremented sequencer id is multi-casted to all 4 RMS

4)Test Cases for Failure Free Sequencer (Vishwas Tomar)

A) 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.

B) 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.

C) 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.

Pseudo Code

Front End

```
// Define message format
String messageFormat =
"Sequenceid;FrontIpAddress;MessageType;function(addEvent,...);userID;newEventID;newEventT
ype;oldEventID;oldEventType;bookingCapacity";


// Define timeout parameters
int timeout = 10000; // 10 seconds
int numRetries = 3;


// Define multicast group for notifying RMs
String multicastGroup = "230.1.1.10";


// Serialize request from client and send to sequencer
String serializedRequest = serializeRequest(clientRequest);
String sequencerResponse = sendToSequencer(serializedRequest);


// Parse sequence ID from sequencer response
int sequenceId = parseSequenceId(sequencerResponse);


// Wait for responses from RMs
List<String> rmResponses = new ArrayList<>();
for (RM rm : RMs) {
  String response = null;
  int numAttempts = 0;
  while (response == null && numAttempts < numRetries) {
    // Send request to RM and wait for response
    response = sendToRM(rm, serializedRequest, timeout);
```

```
    numAttempts++;
  }
  rmResponses.add(response);
}

// Parse RM responses and detect bugs/crashes
List<RmResponse> parsedResponses = new ArrayList<>();
for (String rmResponse : rmResponses) {
  parsedResponses.add(parseResponse(rmResponse));
}
int numCrashes = 0;
int numBugs = 0;
RmResponse majorityResponse = getMajorityResponse(parsedResponses);
for (RmResponse parsedResponse : parsedResponses) {
  if (!parsedResponse.equals(majorityResponse)) {
    // Count as bug or crash depending on type of mismatch
    if (parsedResponse.isCrashed()) {
      numCrashes++;
      notifyRM(multicastGroup, parsedResponse.getRmId(), "23");
    } else {
      numBugs++;
      notifyRM(multicastGroup, parsedResponse.getRmId(), "13");
    }
  }
}
// Send response to client
CommonOutput output = createOutput(numCrashes, numBugs, majorityResponse);
String serializedOutput = serializeOutput(output);
sendToClient(clientIpAddress, serializedOutput);
```

Replica Manager

```
// initialize the RM
sequence_id = 0
queue = create_priority_queue()
hash_map = create_hash_map()

// loop to receive and process messages
while true:
    // receive a message from the sequencer
    message = receive_message_from_sequencer()

    // multicast the message to other RMs
    multicast_message_to_RMs(message)

    // add the message to the queue and hashmap
    add_message_to_queue_and_hashmap(message, queue, hash_map)

// separate thread to execute messages from the queue
function execute_messages():
    while true:
        // get the highest-priority message from the queue
        message = get_highest_priority_message(queue)

        // execute the message on the servers
        execute_message_on_servers(message)
```

```
        // remove the message from the hashmap
        remove_message_from_hashmap(message, hash_map)

// function to handle lost messages
function handle_lost_message(sequence_id):
    // ask other RMs to update their hashmap for the missing sequence ID
    multicast_request_for_missing_sequence_id(sequence_id)

    // wait for responses from other RMs
    responses = wait_for_responses_from_other_RMs()

    // update the local hashmap with the missing messages
    update_local_hashmap_with_missing_messages(responses)
```