Mini Project 1 Report: Server Redundancy Management System (SRMS)

COMP370 AB1 – Software Engineering

Team Name : Tech Titans

Sahijdeep Waraich |Sahijdeep.Waraich@student.ufv.ca | 300209191(Leader)| Step 1: Contributed to requirements gathering and specification. Step 2: Drew UML diagrams on software (Class and Sequence Diagrams). Step 3: Designed the system architecture using OOP principles, drew new UML (encapsulation, single responsibility, inheritance). Step 4: Implemented networking and concurrency. Step 7: Implemented logging, monitoring, and admin interface. Step 8: Conducted testing for normal operation and primary server crash scenarios. State Replication and Election Algorithm. Design Rationale and Class Responsibilities. Project Objectives and How the Solution Addresses Them. Main Functions Implemented. Challenges and Solutions. Made the scripts; run.sh, kill-primary.sh, delay-heartbeat.sh. Prepared and compiled the project report.

Ryan Austin |Ryan.Austin@student.ufv.ca| 300211146| Step 1: Contributed to requirements gathering and specification. Step 2: Assisted in UML diagrams (class diagram).Step 5: Implemented failover and deterministic leader election unique server ID handling, heartbeat tracking, and promotion mechanism. Step 6: Enhanced client failover handling (automatic reconnection, GET_PRIMARY queries, processing requests after promotion). Made the README.md

Bhavika Sachdeva |Bhavika.Sachdeva@student.ufv.ca| 300226894| Step 1: Contributed to requirements gathering and specification. Step 2: Assisted in UML diagrams (class and sequence diagrams). Step 8: Conducted testing for Backup, Simultaneous, Network Delay Simulation, and Recovery. Lessons Learned and Future Improvements.

Easton Abel |Easton.Abel@student.ufv.ca| 300182405| Step 1: Contributed to requirements gathering and specification. Step 5: Implemented failover and deterministic leader election logic with Ryan.

Tanishka Sachdeva |Tanishka.Sachdeva@student.ufv.ca| 300200109| Step 1: Contributed to requirements gathering and specification.

Project Objectives and How the Solution Addresses Them:(Sahij)

The goal of this mini-project was to build a Server Redundancy Management System (SRMS) in Java that demonstrates high availability, reliability, and fault tolerance. Our system simulates a small server cluster with one primary server, multiple backup servers, a monitor, and a client.

Our solution meets the objectives by:

- Reliable service: The primary server handles all client requests. If it fails, a backup is promoted automatically so clients always get responses.
- Fault tolerance: Backups monitor the primary using heartbeat messages. The monitor detects failure and triggers deterministic failover (lowest ID wins).
- Observability and control: A central Logger records events, and the AdminInterface allows monitoring and manual failover.
- Modular, maintainable code: OOP design (encapsulation, single responsibility, inheritance) makes it easy to understand and extend.

Main Functions Implemented:(Sahij)

ServerProcess (abstract)
Base class for all servers. Handles starting/stopping servers, running the main loop, and accepting network connections. Encapsulates common behavior for PrimaryServer and BackupServer.
PrimaryServer
Extends ServerProcess. Handles client requests, replicates state to backups, and logs important events via Logger.
BackupServer

Extends ServerProcess. Monitors the primary server using HeartbeatSender and promotes itself to primary when needed. Logs promotions and state changes.

Monitor

Tracks all servers heartbeat timestamps. Detects primary failure and triggers deterministic failover (backup with lowest ID). Logs key events and prints monitoring info to the console.

Client

Queries the monitor to find the current primary (GET_PRIMARY), sends PROCESS requests, and automatically reconnects if the primary fails. Handles responses from promoted backups.

HeartbeatSender

Runs in its own thread to send regular heartbeat messages from servers to the monitor, ensuring the monitor knows which servers are alive.

ConnectionHandler

Manages incoming socket connections to servers. Ensures that backups process requests only after promotion, while the primary handles requests normally.

Logger

Centralized logging class with timestamps. Used by servers, monitor, and client to record events such as startup, heartbeat, failover, promotions, and client requests.

AdminInterface

Command-line tool for administrators to view logs, check server health, and manually trigger failover if needed.

Main

Entry point of the system. Starts the monitor, server instances (primary and backups), and client simulation. Orchestrates the full SRMS workflow.

Design Rationale and Class Responsibilities:(Sahij)

- Encapsulation: ServerProcess hides internal server details; public methods provide safe access.
- Single Responsibility: Each class handles one main job (networking, heartbeat, request processing, monitoring, logging).
- Inheritance: PrimaryServer and BackupServer extend ServerProcess to share common functionality.
- Failover Logic: Monitor selects the backup with the lowest ID to avoid split-brain.
- State Handling: Simple state replication via heartbeat and monitoring ensures backups are ready to take over.

Step 1: Requirements Gathering and Specification

Build a simulation of a server system with one primary server and multiple backup servers. The client always communicates with the primary. If the primary fails, a backup automatically takes over so service continues without interruption. The system demonstrates server redundancy by ensuring reliability, availability, and minimal downtime. (Easton, Ryan, Sahij, Bhavika, Tanishka)

Stakeholders and Roles

- Primary Server: Handles all client requests, sends heartbeat messages, and ensures the service is available. (Easton)
- Backup Servers: Monitor the primary server and promote themselves if the primary fails. (Easton, Tanishka)
- Monitor/Manager: Tracks heartbeat messages from all servers, detects primary failure, and triggers failover. (Easton)
- System Admin: Observes server logs, monitors server health, and can trigger manual failover. (Sahij, Tanishka)
- Client/User: Sends requests to the primary server and reconnects automatically after failover. (Sahij, Ryan)
- Developers (Team): Implement SRMS, maintain modular and testable code, and handle debugging. (Sahij, Bhavika, Ryan, Easton, Tanishka)

Functional Requirements (FRs):

- FR1: Client can send requests to the current primary server and receive responses. (Ryan, Sahij)
- FR2: PrimaryServer sends heartbeat messages periodically to the Monitor. (Easton)
- FR3: BackupServer monitors primary via heartbeats and promotes itself if the primary fails. (Easton, Tanishka)
- FR4: Automatic failover ensures the client reconnects to the new primary. (Ryan, Sahij)
- FR5: Log events such as server start/stop, heartbeat, failover, and client requests. (Bhavika)
  FR6: Servers can start/stop independently. (Sahij)
- FR7: Multiple backups monitor the primary concurrently. (Sahij)

Non-Functional Requirements (NFRs):

- NFR1: Detect primary failure within 5 missed heartbeats. (Easton, Ryan, Sahij)
- NFR2: Failover completes within 2 seconds. (Easton, Ryan)
- NFR3: Modular OOP design with encapsulation, inheritance, and single responsibility. (Sahij)
  NFR4: Logs are clear and timestamped for debugging. (Bhavika)
- NFR5: System continues working if one backup crashes. (Bhavika)
- NFR6: Client requests are not lost during failover. (Bhavika)
  NFR7: Flexible number of servers without affecting performance. (Tanishka)
- NFR8: System guarantees data integrity during failover. (Tanishka)

Step 2: Use Cases and UML **(Sahij, Ryan, & Bhavika - Class Diagram| Sahij, & Bhavika - Sequence Diagrams)** - UML diagram attached in zip

Step 3: System and OOP Design **(Sahij) -**UML diagram attached in zip

For Step 3, I improved our system design using OOP principles like encapsulation, single responsibility, and inheritance, with key changes from Step 2 UML:

- ServerProcess.java: Abstract class holding common server behavior(start/stop, heartbeat). Data is private; access through public methods (encapsulation).
- PrimaryServer.java: Inherits from ServerProcess.Handles client requests, replicates state to backups, logs events via Logger.
- BackupServer.java: Inherits from ServerProcess. Monitors primary and can promote itself if needed, logging promotions.
- Monitor.java: Tracks servers and heartbeat timestamps, detects failures, and triggers failover. Failover now selects the backup with the lowest ID.
- Client.java: Sends requests to primary, receives responses, and reconnects automatically after failover.
- Logger.java: Centralizes timestamped logging for servers and monitor.
- Main.java: Starts servers, monitor, and client simulations.

Key Changes from Step 2 UML:

- Added Logger class for maintainable logging.
- Failover logic moved inside Monitor.java; no separate FailoverManager.
- Method names updated to match implementation: handleRequest(), promote(), detectFailure()..

Step 4: Networking and Concurrency **(Sahij) -** The following classes were created: BackupServer.java, PrimaryServer.java, Client.java, Monitor.java, ConnectionHandler.java, HeartbeatSender.java, Main.java, and Logger.java.

- Each server runs independently on its own thread, communicating over a network using ServerSocket and Socket.
- Client can connect to the primary server, send requests, and receive responses.
- HeartbeatSender periodically sends heartbeats so the monitor can track server status.
- Monitor detects primary server failures and triggers failover by promoting the backup with the lowest ID.
- Logger.java provides consistent, timestamped logging using LocalDateTime and DateTimeFormatter, ensuring clear event tracking.
- Server ports were adjusted to prevent conflicts, allowing multiple servers to run on the same machine.
- The system supports multiple clients and can simulate failover scenarios for testing.

Step 5: Failover and Election Logic **(Easton and Ryan)**
In Step 5, Easton and Ryan extended Step 4 by implementing failover and deterministic leader election.

- Backup servers actively monitor the primary.
- A backup promotes itself to primary if the primary fails.
- Backups handle client requests only after promotion to avoid conflicts.
- Clients automatically reconnect to the current primary.
- ConnectionHandler ensures backups process requests only when promoted.
- Heartbeats update the monitor in real time.
- The monitor detects primary failure using heartbeat timestamps.
- Failover is deterministic: the backup with the lowest ID is promoted.

- Promoted backups begin handling client requests immediately.
- Demonstrates primary failure by stopping its heartbeat.
- The monitor detects failure and promotes a backup automatically.
- Client requests before and after failover verify system functionality.

Ryan's Contributions:

- Unique server IDs enable deterministic selection of a new primary.
- Monitor tracks heartbeat timestamps for all servers.
- Primary failure detection triggers failover when heartbeats exceed a timeout.
- Promotion mechanism via promote() switches a backup to primary.
- HeartbeatSender updates the monitor in real time, ensuring accurate monitoring.

Step 6: Client Failover Handling **(Ryan)**
Ryan enhanced the client system to work seamlessly with the failover mechanism:

- The client no longer assumes a fixed primary server.
- It queries the monitor over the network (GET_PRIMARY) to determine which server is currently acting as the primary.
- Once the primary is identified, the client sends requests (e.g., "PROCESS") to that server and prints the response to the console.
- The client prints both the original server response and any [PROMOTED] responses if the primary was a backup that got promoted.
- If the client fails to connect to the primary (e.g., during a simulated failure), it detects the failure, waits briefly, and queries the monitor again.
  The client automatically reconnects to the new primary and continues sending requests without losing data.

Step 7: Logging, Monitoring, and Admin Interface **(Sahij)**

For Step 7, I implemented logging and monitoring functionality, as well as an admin interface for system observation and manual control. Key contributions include:

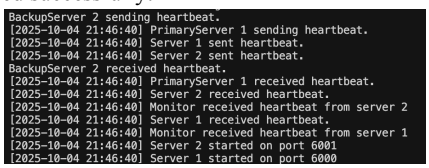
- A centralized Logger class records all important events with timestamps (LocalDateTime and DateTimeFormatter).
- Events logged include server start/stop, heartbeat sent/received, failover, promotion of backups, and client request handling.
- The Monitor class tracks heartbeat timestamps for all servers.
- It detects primary server failure if no heartbeat is received within the threshold (timeoutThresholdMs).
- Failover is triggered automatically, promoting the backup with the lowest server ID.
- Important monitoring events are printed to the console and logged, making system status observable in real time.
- The AdminInterface class provides a command-line tool for administrators.
- The logging and admin interface ensures the system is observable, debuggable, and manageable.
- Admins and developers can track system behavior during normal operation or failure scenarios.
- The combination of automated logging, real-time monitoring, and manual control tools improves maintainability and reliability.

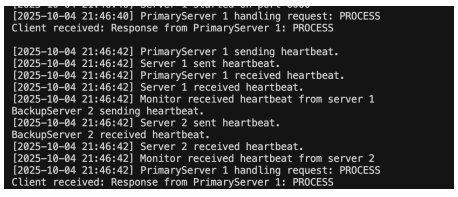
Step 8: Testing and Failure Scenario

i) *Normal operation:*(Sahij) At [2025-10-04 21:46:40], the monitor successfully registered Server 1 as the Primary and Server 2 as a Backup. Both servers started sending and receiving heartbeat messages, indicating that the system was functioning normally:

Clients were able to send requests to the primary server, and responses were returned successfully:

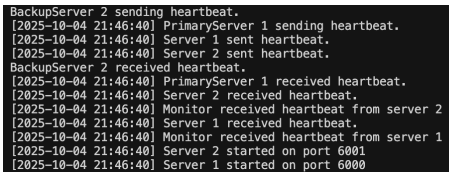
- Both servers were active, with the primary handling client requests and the backup sending heartbeats to the monitor.
- No failures occurred during this period, so the failover logic was not triggered.





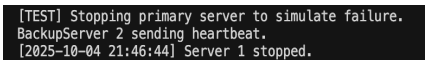
- The system met the functional requirement that a client request is served by the primary and the backup server remained ready.
- Heartbeats continued regularly, ensuring that the monitor could detect failures if they occurred.

ii) *Primary Crash:*(Sahij) At [2025-10-04 21:46:40],
the monitor successfully registered Server 1 and Server 2,
with Server 1 designated as the Primary and Server 2 as a Backup.
Both servers started sending and receiving heartbeat messages normally:

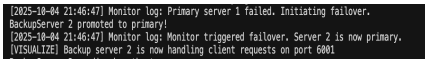


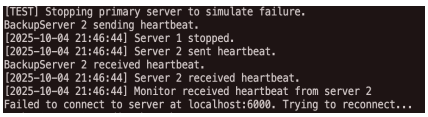
Clients were able to send requests to the primary,
and Server 1 responded correctly:

At [2025-10-04 21:46:44], the primary
server was intentionally stopped to simulate a failure:



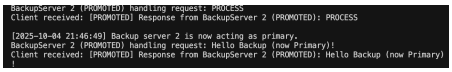
Immediately after the crash, heartbeat messages from the
primary stopped, and the client detected that the primary was unavailable:





The backup server remained active, continuing to send heartbeats, while
the monitor detected the failure of the primary at [2025-10-04 21:46:47]:

Once promoted, the backup server successfully handled client requests
as the new primary:

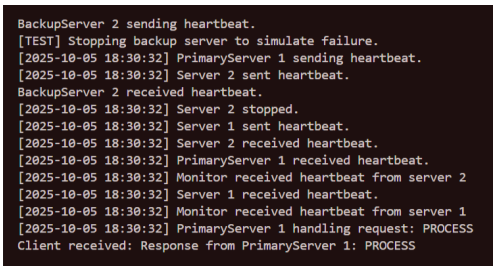


- The monitor detected the primary failure approximately 3 seconds after the server was stopped.
- Failover was automatic: BackupServer 2 was promoted to primary and continued handling client requests without errors.
- The client correctly attempted reconnection and successfully communicated with the new primary.
- Heartbeats ensured the monitor could reliably determine server health, confirming system resilience.

iii) *Backup crash:*(Bhavika)

At [2025-10-05 18:30:32], the monitor successfully registered Server 1 as the Primary and Server 2 as the Backup. Both servers initially exchanged heartbeat messages, indicating normal operation. To simulate a failure, the Backup server was stopped manually.

Immediately after shutdown, heartbeat messages from Server 2 stopped, but Server 1 continued to send and receive heartbeats normally. The monitor maintained contact with the primary and did not trigger any failover, confirming that the system can tolerate a backup failure without affecting client service.



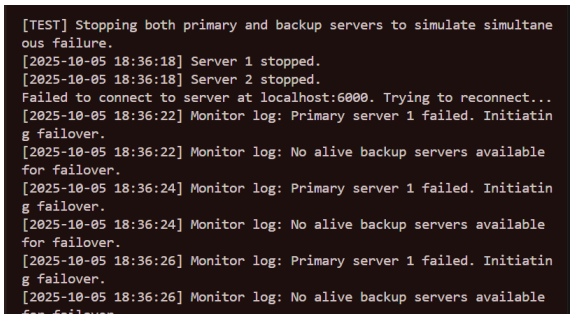
- Event Time: 18:30:32 – Backup stopped
- Detection Time: ≈ 2 seconds (monitor timeout)
- Failover Triggered: No (primary still active)
- System Status: Stable, primary operational
- Client Response: Response from PrimaryServer 1: PROCESS

iv) *Simultaneous failures:*(Bhavika)

At [2025-10-05 18:36:18], both the Primary (Server 1) and Backup (Server 2) were intentionally stopped to simulate a complete system failure. Immediately after the shutdown, the monitor detected that no heartbeat messages were being received from either server.

At [2025-10-05 18:36:22], the monitor initiated a failover procedure but found no active backup servers, logging the message "No alive backup servers available for failover." The system entered an unavailable state, and the client continuously retried connections without success.

- Event Time: 18:36:18 – Both servers stopped
- Detection Time: ≈ 4 seconds
- Failover Time: Not applicable (no backup available)
- System Status: Unavailable – All servers down
- Client Response: Connection retries failed until restart

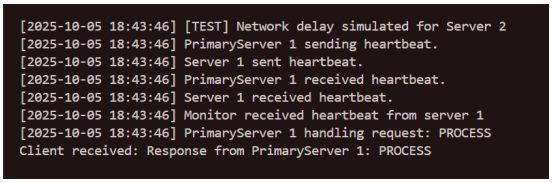


v) *Network delay simulation:*(Bhavika)

At [2025-10-05 18:43:46], a deliberate delay was introduced in the heartbeat of Server 2 (Backup) to simulate a network lag scenario.
While the backup's heartbeat was delayed, the primary server (Server 1) continued to send and receive its heartbeats normally, and clients still received correct responses such as "Response from PrimaryServer 1: PROCESS."

Observations:

- The monitor kept receiving regular heartbeats from the primary server, proving that the system remained stable even when one server experienced latency.
- The artificial delay did not trigger a false failover, confirming that the system's timeout and detection thresholds were correctly tuned.
- This test verified that minor network delays are tolerated without interrupting normal client-server communication.

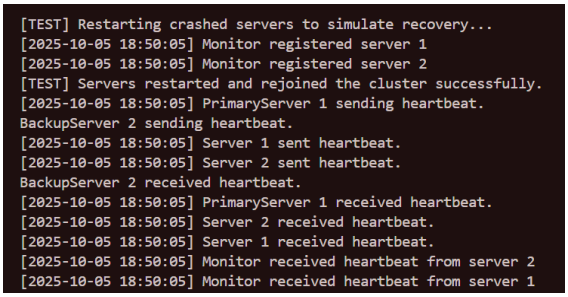


vi) *Recovery:*(Bhavika)

At [2025-10-05 18:50:05], both the primary and backup servers were manually restarted to simulate system recovery after a complete failure. Once restarted, the monitor successfully re-registered both servers and began receiving heartbeat messages from them again, confirming that they had rejoined the cluster.

Observations:

- The monitor correctly detected and registered Server 1 and Server 2 after restart.
- Heartbeats from both servers resumed immediately, indicating that communication with the monitor was re-established.
- Clients could again send requests to the primary server and receive valid responses without errors.



State Replication and Election Algorithm: (Sahij)

- *Heartbeat-Based Monitoring*: Each server periodically sends heartbeat messages to the Monitor, which records timestamps to track server liveness. Backups maintain a ready state to take over if the primary fails.
- *Election Algorithm*: Deterministic failover logic is implemented in the Monitor class. When the primary server fails (heartbeat timeout), the monitor promotes the alive backup with the lowest server ID. Only one backup is promoted at a time, preventing conflicts.

Challenges and Solutions: (Sahij)

- *Concurrent Server Execution*: Running multiple servers (primary and backups) along with the monitor on the same computer initially caused port conflicts. To fix this, we assigned unique ports: 6000 for the primary, 6001/6002 for backups, and 7100 for the monitor. This made sure all processes could run at the same time without connection errors.
- *Reliable Failover*: At first, the monitor sometimes promoted backups too early or too late because of heartbeat timing issues. We added a 5-second timeout in Monitor.detectFailure() to check the last heartbeat from the primary. If no heartbeat is received within this limit, Monitor.triggerFailover() promotes a backup. This made failover detection stable and predictable. For example, when the primary server is stopped in Main.java, the monitor now waits 5 seconds before promoting the backup, avoiding false failovers.
- *Promoted Backup Request Handling*: Before, promoted backups didn't process client requests correctly. We updated BackupServer.handleRequest() and ConnectionHandler.run() so that promoted backups now handle requests like a primary, while unpromoted backups reject them. For instance, after stopping the primary, the client request "PROCESS" successfully reaches the promoted backup and gets a proper response.
- *Heartbeat Logging and Monitoring*: We added detailed logs for both sending and receiving heartbeats using the centralized Logger class. This lets us track server status and failover events in real-time.
- *Thread-Safe Heartbeat Updates*: Multiple heartbeat threads update timestamps in Monitor.recordHeartbeat(). Synchronizing this method prevents race conditions that could trigger incorrect failovers.
- *Admin Interface Usability*: We implemented an AdminInterface with status and failover commands. This allows manual inspection and testing of server states. For example, typing status shows which servers are running, stopped, or promoted.
- *Error Handling for Socket Connections*: We improved exception handling in Client.sendRequest() and ConnectionHandler.run(). This prevents server crashes when clients disconnect unexpectedly or the network is slow.
- *Client Reconnection*: Originally, clients only printed a connection error when the primary failed. We updated the client loop in Main.java to continuously query the monitor using GET_PRIMARY and retry until it connects to the new primary. This keeps the system running smoothly during failovers.
- *Logging & Monitoring*: All servers and the monitor now log events with timestamps using the Logger class. Combined with the AdminInterface, this makes it easy to debug and watch the cluster in real-time.
- *Independent Execution for Grader Scripts*: We added main() methods to PrimaryServer, BackupServer, Monitor, and AdminInterface. Each can be run independently via scripts (run.sh) even though the simulation is mainly driven by Main.java.
- *Startup and Race Conditions*: Servers and the monitor run in separate threads, which sometimes caused the monitor to check heartbeats before servers were fully initialized. We fixed this by synchronizing server registration in Monitor.registerServer() and adding Thread.sleep() delays during startup.
- *Failure Simulation Scripts*: Provided scripts to simulate failures for testing: kill-primary.sh - stops the JVM hosting the primary server. delay-heartbeat.sh - simulates delayed heartbeat handling to test failover robustness.
- *Duplicate Backup Processes*: While testing, we noticed multiple BackupServer instances running simultaneously (seen via jps). This caused heartbeat confusion and "No primary server running" errors. We fixed it by cleaning up old background processes before rerunning run.sh.
- *Heartbeat Delay Script Issue*: Initially, delay-heartbeat.sh failed with the message *"no primary server running"*. The issue was due to monitor timing. We corrected it by ensuring the script targeted an existing backup process and verified that Monitor still detected valid heartbeats afterward.

Lessons Learned and Future Improvements: (Bhavika)

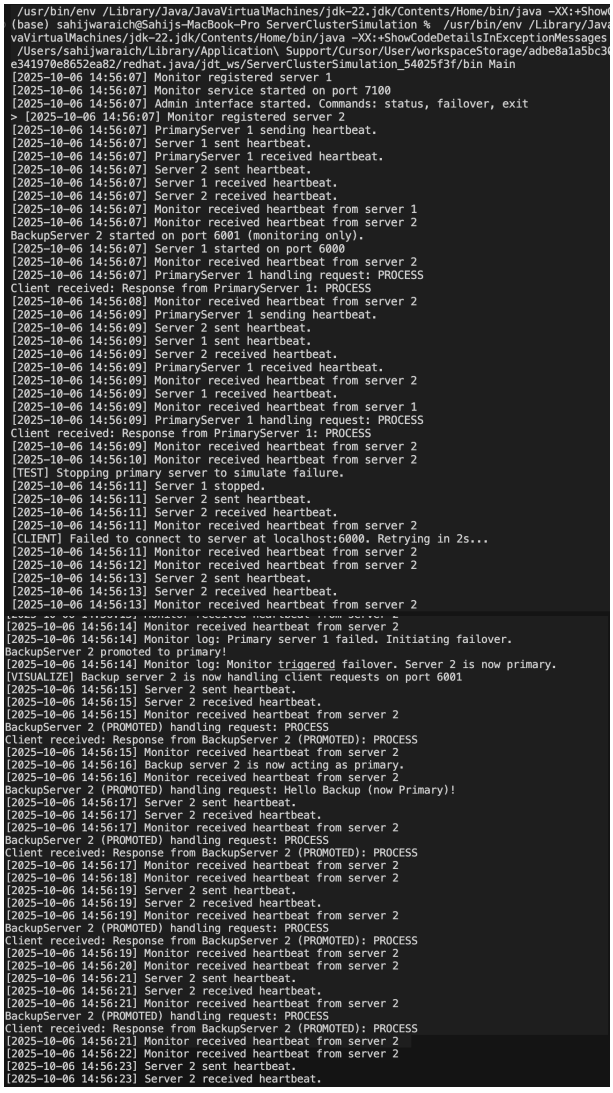
- During this project, we learned how the real system handles failures and keeps running smoothly.
- We got hands-on experience with Java sockets, threads, and server-client communication.
- Setting up heartbeats and failover gave us a better understanding of how monitoring and redundancy work in cloud systems.
- We also realized how important timing and synchronization are – even a small delay can change how the system behaves.

Overall, we learned how to turn software engineering concepts like OOP design, concurrency, and reliability into a working system.

In the future, we would like to make the system more advanced by adding:

- A Graphical User Interface (GUI) dashboard to show live server status, heartbeats, and logs in real time.
- A database or log file to save all server events and failure history for better analysis
- More backup servers to handle multiple simultaneous failures.
- An automatic notification system that alerts the administrator (via email or message) when a failure occurs.
- A more detailed state synchronization feature so all backups always have the latest system data

Output: (Sahij)



Running and Reproducing: (Sahij)