Mini Project 1 Specification:
Server Redundancy Management System (Java)

COMP370 - Software Engineering

Project Motivation

Software systems that provide services to users must be reliable and available. In many real-world settings—cloud services, financial transaction systems, telecommunication control planes—downtime is costly, or even dangerous. To achieve high availability, systems often employ server redundancy: multiple server instances cooperate so that if one server fails, others can take over the workload with minimal disruption. Designing such systems requires careful application of software engineering principles: requirements engineering to define what "available" means for stakeholders; object-oriented design to build modular, testable components; UML modeling to capture architecture and behaviour; and networked, concurrent programming to implement distributed interactions.

This mini-project asks students to design, implement, and evaluate a **Server Redundancy Management System** (SRMS) in Java. The SRMS will simulate multiple server processes (each running as a separate JVM program), a monitor/manager that tracks server health, primary-backup failover logic, and a client component that issues control or test requests. The project integrates topics from lectures: requirements engineering, OOP, UML, client-server architecture, threading/-concurrency, and quality attributes (reliability, maintainability, performance, testability).

High-level Project Description

Your SRMS must simulate a small cluster of servers that provide a simple service (for example: accept and respond to a basic 'PING' or 'PROCESS' request). Features required:

1. **Multiple server processes**: Implement at least three independent server processes (Java programs) that listen on different ports and can be started/stopped independently.
2. **Primary–Backup architecture**: At any time one server is the *Primary* (handles client requests), others are *Backups* (standby). Backup servers monitor the primary via heartbeats.
3. **Heartbeat / health checks**: Servers send periodic heartbeat messages to a monitor and/or to each other. The monitor detects failure if heartbeats stop.
4. **Failover (automatic)**: When the primary fails, a backup detects the failure and performs failover to become primary. The failover must avoid split-brain where possible (describe and handle simple split-brain avoidance).
5. **Client interface**: A client program that sends requests (e.g., 'PROCESS job-id') to the primary. If the primary is down, the client should be able to reconnect to the new primary (discoverable via the monitor).

6. **Logging and monitoring**: Servers and monitor log events with timestamps. Include a simple console or file log describing state changes and failovers.
7. **Simulated failure scenarios**: Provide scripts or commands to simulate failures (kill process, freeze process), network delays, or partitions (can be simulated with the server ignoring heartbeats).

Learning Objectives

By completing the project students will:

- Apply requirements engineering: specify functional and non-functional requirements and map them to design choices.
- Use UML (use case, class, sequence diagrams) to plan the system.
- Architect and implement a client-server distributed system in Java with threads and sockets.
- Implement primary-backup failover and simple state replication.
- Design and execute test cases for reliability, failover and recovery scenarios.
- Practice collaborative development, code organization, and documentation.

If you face any challenges:

1. Do your own research!
2. Ask your own team members.
3. Ask question during lecture sessions.
4. Feel free to use `https://chatgpt.com/`, `https://gemini.google.com/app`, `https://claude.ai/new`, etc. **BUT** always double check their outputs and ask **WHY**.

Detailed Steps (what teams must do)

Below is a step-by-step roadmap. Each step must be documented in your project report and in-code comments where appropriate.

Step 1: Requirements Gathering and Specification

Here is the list of tasks that need to be performed in this step:

- Identify stakeholders, e.g., system admin, end-user, service consumers.
- Write a short problem statement (1–2 sentences).
- Produce a list of functional requirements (FR) and non-functional/quality requirements (NFR).

Minimum Functional Requirements (examples)

1. FR1: The system shall allow a client to send a simple request and receive a valid response from the current primary.
2. FR2: The system shall elect a backup to become primary if the current primary fails.

Minimum Non-Functional Requirements (examples

- NFR1: The system shall detect primary failure within 5 heartbeats.
- NFR2: The system must resume serving requests within 2 seconds after failover (simulated environment).

Example Problem Statement

> Build a simulation of a server cluster with a primary node and multiple backup nodes, such that client requests are served by the primary and backups can take over automatically and consistently when failures occur.

Step 2: Use Cases and UML

Create UML artifacts (you may use draw.io, Lucidchart, or hand-drawn diagrams). Here is just an example, and you might want to consider different artifacts:

1. **Class Diagram**: Show classes and interfaces such as:
   - `ServerProcess` (abstract)
   - `PrimaryServer` (extends `ServerProcess`)
   - `BackupServer` (extends `ServerProcess`)
   - `Monitor`
   - `Client`
2. **Sequence Diagrams**: You need to create at least two sequence diagrams to show how the system behaves in important scenarios:
   (a) **Startup and Leader Election:** Illustrate the step-by-step interactions between servers when the system starts. Show how the primary server is elected among the available servers and how they coordinate to establish roles. Include messages exchanged and decisions made during this process.
   (b) **Client Request Flow:** Create two related diagrams here:
   - How a client's request is handled during normal operation when the primary server is active.
   - How the system handles a client request during a failover event, when the backup server takes over because the primary has failed.

   These diagrams should show the sequence of messages between the client, primary, and backup servers, highlighting any differences in the flow.

Step 3: System and OOP Design

Design classes with clear responsibilities. Here is just an example, and you might want to consider different OOP design:

- `IMessageSerializer`: interface for message (de)serialization.

- `ServerProcess`: encapsulates server socket, thread loop, message handlers, and health-check logic.
- `FailoverManager`: decides which backup becomes primary (e.g., deterministic: lowest ID).

Here is the list of the Design principles that you should follow.

- **Encapsulation:** Keep all internal details of the server hidden from other parts of the program. Do not allow direct access to server data or internal methods. Instead, provide a clear, simple set of public functions (APIs) that other parts can use to interact with the server safely and consistently.
- **Single Responsibility:** Each class should focus on one specific job or responsibility. For example, one class might handle connection management, another handles request processing. This makes your code easier to understand, test, and maintain because changes in one class won't affect unrelated functionality.
- **Use inheritance cautiously:** Identify the common behaviors that all servers share and put them into an abstract base class that cannot be instantiated by itself. Then create subclasses like `PrimaryServer` and `BackupServer` that inherit from the base class and implement their specific differences.

Step 4: Networking and Concurrency Implementation

Each server process is an independent Java application. Use `ServerSocket` to accept incoming connections (from clients or other servers). Here is the simplified version of the server main loop and Heartbeat mechanism that you can develop. Of course if you have your own implementation, it is also fine!

Server main loop (simplified)

```
public class ServerProcess {
  private ServerSocket serverSocket;
  private volatile boolean running = true;
  private ExecutorService threadPool = Executors.newCachedThreadPool();

  public void start(int port) throws IOException {
    serverSocket = new ServerSocket(port);
    while (running) {
      Socket s = serverSocket.accept();
      threadPool.submit(new ConnectionHandler(s));
    }
  }
  public void stop() throws IOException {
    running = false;
    serverSocket.close();
    threadPool.shutdownNow();
  }
}
```

Heartbeat mechanism (simplified)

```
public class HeartbeatSender implements Runnable {
  private final String monitorHost;
  private final int monitorPort;
  private final String serverId;
```

```
  public void run() {
    while (true) {
      sendHeartbeat();
      Thread.sleep(heartbeatIntervalMs);
    }
  }
}
```

Step 5: Failover Logic and Election

Now implement the Failover and election logic using a simple deterministic approach as follows:

- Assign each server a unique identifier (ID), such as a numeric value given when the server starts. This ID will be used to compare and select servers during election.
- The monitor component keeps track of all the registered servers along with the timestamp of their most recent heartbeat messages. Heartbeats are periodic signals sent by servers to indicate they are alive.
- If the monitor notices that the primary server's heartbeat has not been received within a certain timeout threshold, it assumes the primary server has failed. At this point, the monitor initiates the failover process.
- The failover process involves selecting a new primary server from the available backups. To do this deterministically, the monitor chooses the backup server with the lowest ID among those currently alive.
- Once a new primary is chosen, the monitor sends a `PROMOTE` message to that backup server. Upon receiving this message, the backup server promotes itself to primary and begins handling client requests.

Step 6: Client Implementation and Discovery

In this step, you will implement the client-side logic to communicate with the server system. Your client should:

- **Query the monitor service:** The client must ask the monitor to find out which server is currently acting as the primary. The monitor will respond with the primary server's address (host and port).
- **Send requests to the primary:** After discovering the primary, the client should send a 'PROCESS' request to the primary server and then display whatever response it receives.
- **Handle connection failures:** If the client's connection to the primary server fails (for example, if the server crashes or is unreachable), the client should detect this failure. Upon failure, the client must re-query the monitor to get the updated primary server information and try connecting again. This process should repeat until a successful connection is made.

Client reconnection logic pseudocode

```
while (!connected) {
  // Ask the monitor for the current primary server's address
  primary = monitor.getPrimary();
  try {
```

```
    // Attempt to connect to the primary server
    connect(primary.host, primary.port);
    connected = true; // Connection successful
  } catch (IOException e) {
    // If connection fails, wait before retrying
    Thread.sleep(1000);
  }
}
```

This loop ensures the client keeps trying to connect to the primary server, updating its knowledge if the primary changes.

Step 7: Logging, Monitoring, and Admin Interface

In this step, you need to implement mechanisms to record important events happening in the server and provide tools to monitor the server's health and status. Specifically:

- **Logging:** Your server should keep detailed logs of key events to help understand what is happening during execution and to assist with debugging or troubleshooting. Each log entry must include a timestamp (the exact time when the event happened). The logs should record the following events:
  - When the server starts up or shuts down.
  - When a heartbeat message is sent or received (to confirm the server is alive).
  - When failover occurs (for example, when the backup server takes over because the primary fails).
  - When the server applies or acknowledges an update to its internal state.
  - When client requests are received and processed.
- **Monitoring:** Use the logged information to monitor the server's status and performance. This can be as simple as printing key status messages to the console or more advanced by creating a dashboard or admin interface.
- **Admin Interface:** Provide an interface (command-line or graphical) that allows an administrator to:
  - View recent logs and status updates.
  - Check server health (e.g., confirm if the server is running and responsive).
  - Trigger manual failover or other administrative actions if needed.

  This step is important to ensure your system is observable and manageable during testing and deployment.

Step 8: Testing and Failure Scenarios

In this step, your team must design, execute, and document a series of test scenarios to validate the behavior and robustness of your server system under different conditions. For each scenario, carefully observe and record the system's response to ensure it behaves as expected.

Scenario	Description and Tasks
Normal operation	Start the monitor and 3 server instances. Confirm that a primary server is correctly selected. Start client(s) and send requests to the system. Verify clients receive correct responses and that the server state is properly replicated to backups.
Primary crash	Forcefully stop (kill) the primary server process. Check that the monitor detects this failure promptly and promotes a backup server to primary. Verify that clients reconnect automatically and continue to have their requests served without errors.
Backup crash	Kill one of the backup servers during operation. Verify the primary server continues serving client requests without disruption. Restart the killed backup server and confirm it synchronizes its state correctly with the primary before resuming normal operation.
Simultaneous failures	Kill the primary server and one backup server at the same time. Observe if the cluster still successfully elects a new primary server, if possible. Document any system limits encountered (e.g., if all servers fail, what happens?).
Network delay simulation	Introduce artificial delays in heartbeat messages or their handling (for example, by adding sleep statements in the code). Verify that your failure detection thresholds are tuned to avoid false failover triggers caused by delays.
Recovery	Restart all crashed servers. Observe the process of servers rejoining the cluster, synchronizing their state, and being assigned correct roles (primary or backup).

For each scenario, your team must record the following details in your documentation:

- **Time of event:** Record the exact timestamp when the failure or event occurred.
- **Detection time:** Measure how long it took for the monitor or system to detect the primary server failure.
- **Failover time:** Record the time from failure detection to when the new primary starts accepting client requests.
- **State consistency:** Note any inconsistencies in the server state observed during the scenario, and describe how your system handled or resolved them.

Step 10: Documentation and Report

Your report (max 5 pages) must include:

- Team information and responsibilities.
- Requirements (FR/NFR) and traceability to design/implementation.
- UML artifacts (embedded or attached).
- Design rationale and class responsibilities.
- Description of state replication protocol and election algorithm.

- Test scenarios and results (including timestamps and explanations).
- Lessons learned and future improvements.

Evaluation Criteria and Expectations

Functional Correctness (40%)

- Correct implementation of primary-backup behavior.
- Heartbeat detection
- promotion logic
- Clients can discover the primary and resume after failover.

Design and UML (20%)

- Well-formed use cases, class and sequence diagrams that match implementation.
- Clean OOP: appropriate use of classes, interfaces, and inheritance.

Testing and Reliability (10%)

- Comprehensive test scenarios executed and documented.
- Evidence of handling edge cases and recovery.

Code Quality and Documentation (30%)

- Code readability, comments, and modularity.
- Clear README and instructions to run the simulation.
- Project report quality (concise, clear, max 3 pages).

Running and Reproducing (instructions for graders)

1. Provide a top-level `run.sh` (or `run.bat`) script to start: monitor, server1, server2, server3, admin client.
2. Provide scripts to simulate failures:
   - `kill-primary.sh` — kills the JVM hosting the current primary.
   - `delay-heartbeat.sh` — triggers a delay in heartbeat handling.
3. Provide clear README: how to compile (‘javac‘ or Maven), ports used, and how to reproduce test scenarios.

Deliverables

Your team must submit the following items by the deadline:

- **java source files** containing your implementation including all functions.

- A **project report** (PDF, max 5 pages) which includes:
  - Information of your team members including (first name, last name, student ID and email address)
  - An explanation of the project objectives and how your solution addresses them.
  - Descriptions of the main functions you implemented and how they work.
  - A discussion of the challenges your team faced during the project and how you overcame them.
  - A summary of each team member's contributions to the project, demonstrating collaborative effort.
  - Only one submission per team is required. Please submit a single ZIP file named `COMP370-MINI-PROJECT-1.zip` via `https://brightspace.ufv.ca/d2l/home` containing all the required files.
  - Late submissions will not be graded unless an extension has been granted. Requests for extensions must be made at least **one week** before the submission deadline.

This report helps us understand your approach, problem-solving process, and teamwork (all critical aspects of software development beyond writing code).

Grading and Submission Notes

- Only one ZIP per team. The ZIP must contain:
  - `src/` folder with all Java sources.
  - `scripts/` folder with run/kill/simulate scripts.
  - `README.md` with run instructions and port numbers.
  - `report.pdf` (max 5 pages).
  - UML diagrams (PNG/PDF) in `docs/`.
  - Unit tests (if any) in `tests/`.
- Academic honesty: all code must be your team's original work. If you use public libraries or code snippets, cite them in the report.

Hints, Tips and Common Pitfalls

- Start small: implement a single server and a client first.
- Implement the monitor early — it simplifies failover testing.
- Use threads carefully!
- Log extensively during development to understand timing issues.
- Keep message formats simple.

- When in doubt about election strategy, choose deterministic rules (lowest ID wins) to simplify reasoning.

Final Remarks

This mini-project is designed to touch many core software engineering concepts: requirements analysis, UML modeling, OOP design, concurrency, networking, and testing. It is intentionally open-ended in some aspects to encourage creativity: teams should document trade-offs, clearly explain assumptions, and justify design choices in the report. Demonstrations of robustness (good logging, clean failover, and recovery) will be rewarded.

Good luck and have fun building a resilient system!