Mini Project 1.

Mike Buss, Bhupinder Singh Gill, Armaan Kandola, Brayden Schneider, Eric Thai

University of the Fraser Valley, School of Computing

COMP-370-ON1: Software Engineering

Majid Babaei

February 13th , 2026

TABLE OF CONTENTS

INTRODUCTION

This project implements a Server Redundancy Management System (SRMS) in Java capable of automatic failover and recovery. The system is based on a simplified Raft-style consensus approach, where servers coordinate leader election, heartbeat monitoring, and replicated logs to maintain availability.

The goal of the project is to apply software engineering principles, including requirements engineering, object-oriented design, UML modeling, networking, and concurrency, to build a system. The implemented system includes a monitor-like coordination mechanism embedded in the Raft servers, a client that discovers the active leader, and a logging framework for observability.

I. BACKGROUND

A.Team Members

Fig 1: Team Member Table

Name	Student ID	Email
Mike Buss(Team Lead)	300066231	Michael.buss@student.ufv.ca
Bhupinder Singh Gill	300140363	bhupindersingh.gill@student.ufv.ca
Armaan Kandola	300209870	Armaan.Kandola@student.ufv.ca
Brayden Schneider	300200395	Brayden.Schneider@student.ufv.ca
Eric Thai	300228515	Cong.thai@student.ufv.ca

B.Work Division

Mike is responsible for diagrams and charts.
Bhupinder and **Brayden** are responsible for the coding and setting up the server.
Armaan will work on the project report document.
Eric is doing the presentation, and assisting overall where needed.

To clarify, while this is the work division, all group mates were required to help where needed so as to not overload any group member with too much work if the scale exceeded expectation, and to ensure a fair workload division.

II. REQUIREMENTS ANALYSIS

A.Problem Statement:

Design and implement a simulation of a distributed server cluster in which peer nodes elect a leader to handle client requests, while follower nodes automatically assume leadership when failures occur, ensuring continuous service availability.

B.Stakeholders:

System administrators who are responsible for monitoring cluster health
Clients who rely on uninterrupted service
Developers who are maintaining the system

C.Functional Requirements

FR1: The system shall elect a leader among server nodes.
(Implemented in RaftNode via startElection(), handleRequestVote(), and becomeLeader())
FR2: The leader shall handle all client job requests.
(Implemented in RaftServer via handleClientConnection() and Client via sendRequest())
FR3: The system shall replicate log entries across nodes.
(Implemented in RaftNode using handleAppendEntries(), appendAsLeader(), nextIndex, and matchIndex)
FR4: The cluster shall automatically recover from leader failure.
(Implemented in RaftNode through election timeouts and RaftServer through heartbeat monitoring and election triggering)
FR5: Clients shall rediscover the new leader after failover.
(Implemented in Client using discoverLeader() and automatic retry logic in sendRequest())
FR6: All major system events shall be logged.
(Implemented in Logger and integrated across RaftNode, RaftServer, and Client)

D.Non-Functional Requirements

NFR1: Failure detection shall occur within a bounded heartbeat timeout.
(Implemented in RaftNode via election timeouts and RaftServer via periodic heartbeat checks)
NFR2: The system shall resume service quickly after failover.
(Implemented in RaftNode through automatic leader election and Client through leader rediscovery and request retry)
NFR3: Logs shall be thread-safe and timestamped.
(Implemented in Logger using synchronized logging and timestamped entries)
NFR4: The architecture shall support modular extensibility.
(Achieved through separation of concerns across RaftNode, RaftServer, Client, and Logger)

III. SYSTEM DESIGN AND IMPLEMENTATION

A.System Overview

The main objectives of this project were to:

1. **Implement a reliable Server Redundancy Management System**
2. **Ensure uninterrupted client service**
3. **Maintain consistent system state**
4. **Provide observability and debugging tools**
5. **Demonstrate core software engineering principles**

Our solution addresses these objectives by using a Raft-based distributed architecture in which multiple nodes elect a leader and replicate system state. The leader node processes all client requests, while backup nodes continuously monitor the leader through heartbeat messages. If the leader has an issue or does not even exist, the nodes automatically initiate an election and promote a new leader without manual intervention. The system follows a leader-follower topology where each node maintains a replicated log and local state machine. Nodes communicate using lightweight TCP messages for heartbeats, vote requests, and log replication, and timers drive election and heartbeat behavior to ensure consistent role transitions.

The system includes a client component that dynamically discovers the current leader and reconnects after failover. Thread-safe logging records system events to enable monitoring, debugging, and analysis of cluster behavior. The architecture intentionally separates concerns:

RaftNode manages consensus state and timers, RaftServer handles networking and client routing, Client manages discovery and retries, and Logger centralizes observability. This separation improves testability, modularity, and future extensibility. Finally, we demonstrate the principles of requirements engineering, OOP, UML, client-server architecture, threading/ concurrency, and quality attributes during the development of this system.

B.Main Functions

RaftNode (RaftNode.java)

RaftNode implements the core consensus logic. It works by managing:

- Leader election and voting (startElection(), handleRequestVote())
- Leader tracking (currentLeaderId)
- Log replication state (nextIndex, matchIndex)
- AppendEntries and RequestVote handling (handleAppendEntries())
- Detailed event logging
- Leader replication helpers (becomeLeader(), appendAsLeader()) for consistent log synchronization and commit tracking.

RaftServer (RaftServer.java)

RaftServer wraps RaftNode with networking infrastructure. It works by/manages:

- TCP server for inter-node communication
- Client request listener
- Heartbeat scheduling
- Majority commit enforcement
- Applying committed log entries to the state machine
- Graceful shutdown handling
- Integrated logging

Logger (Logger.java)

The Logger provides thread-safe timestamped logging to:

- Console output
- Persistent log files
- This improves debugging and monitoring of cluster behavior.

Client (Client.java)

The client works by:

- Discovering the leader using GET_LEADER requests
- Sending PROCESS_JOB requests
- Automatically retrying when leaders change or failover occurs
- Providing an interactive command-line interface

C.Election and Failover Strategy

The cluster uses Raft-style deterministic leader election. When a leader fails, remaining nodes initiate an election and select a new leader based on term and voting consensus. Heartbeats maintain cluster stability and prevent issues such as split-brain scenarios.

Each node uses an election timeout randomized within a bounded range to reduce split votes. When a timeout occurs, a node increments its term, becomes a candidate, and requests votes from peers. A candidate becomes leader only after receiving a majority. Leaders periodically broadcast heartbeats (empty AppendEntries) to assert authority and reset follower timers. If a leader fails, followers stop receiving heartbeats, trigger a new election, and the system converges on a new leader. Term numbers ensure stale leaders step down if they

observe a higher term. This guarantees at most one leader per term and minimizes split-brain behavior.

IV. TESTING SCENARIOS

A.Scenario Testing

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 Introduces 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).

Fig 2: Scenario Test Results Table

Scenario	Time of event	Detection Time	Failover time	State Consistency
Normal operation	Start* 18:10:03	Election Started 18:10:07	First HeartBeat at 18:10:09	Node 1 and Node 2 both were candidates at the same time, Node 1 stepped down and voted for node (which became the leader eventually)
Primary crash	Node 3(leader) crashed at 18:18:53	Node 1 timed-out at 18:18:57 (~4 seconds)	Node 1 started election instantly 18:18:57, and was elected under 10ms.	18:18:59, system is fully stable and node 1 is the new leader. The client was redirected to node 1
Backup crash	Leader was Node 3, 1 and 2	Leader receives a timeout error	No failover, there are still 2	The backup crashes but we

	are followers Backup(2) was killed at 18:23:55	18:26:55	nodes we are ok	are ok, there is still 1 leader and 1 follower (and i tested node 2 restarted and rejoined correctly)
Simultaneous failures	Kill the primary and 1 follower Following raft with 3 nodes it should be impossible to recover.	Node 3 receives its last heartbeat (18:31:51) and goes to start a election, that timeout so it trys again (and so on)	Unrecoverable With 1 node left, node 3 cannot receive enough votes The first election timeout was at 18:31:56	This is correct and expected behavior with 3 nodes, However, small bug node 3 still thinks the leader is node 2 and tells the client even though its dead. This is just a minor correctness issue.
Network delay simulation	This has an unstable leader/ election timeout Caused by a simulated missed heartbeat Node 3 starts election at 18:37:50 and is elected under 10ms	Node 3 misses a heartbeat, and node 2 starts a election at 18:38:05	Node 1, sees nodes 2 election and also hasn't heard from node 3, so it votes for node 2 to be elected and it is (about 30ms)	Node 3 comes back and steps down because node 2 has a higher term We are ok! This could be for many reasons like scheduling delay, network delay, election timeout, leader stalled out but the algorithm react and recover as long as a majority of the nodes are alive and talking
Recovery Restart all crashed servers	Cluster restarts at 18:26:16 By hitting the	Node 1 timeout and starts election 18:26:19	Node 1 was elected in about 35ms 18:26:19	Stable at 18:26:21 About 5.5 second recovery

	hard reset button			time(do note we don't have persistence yet so this is actually just the same as starting, also works with individual nodes)

V. UML DIAGRAMS

Fig 3: Class Diagram

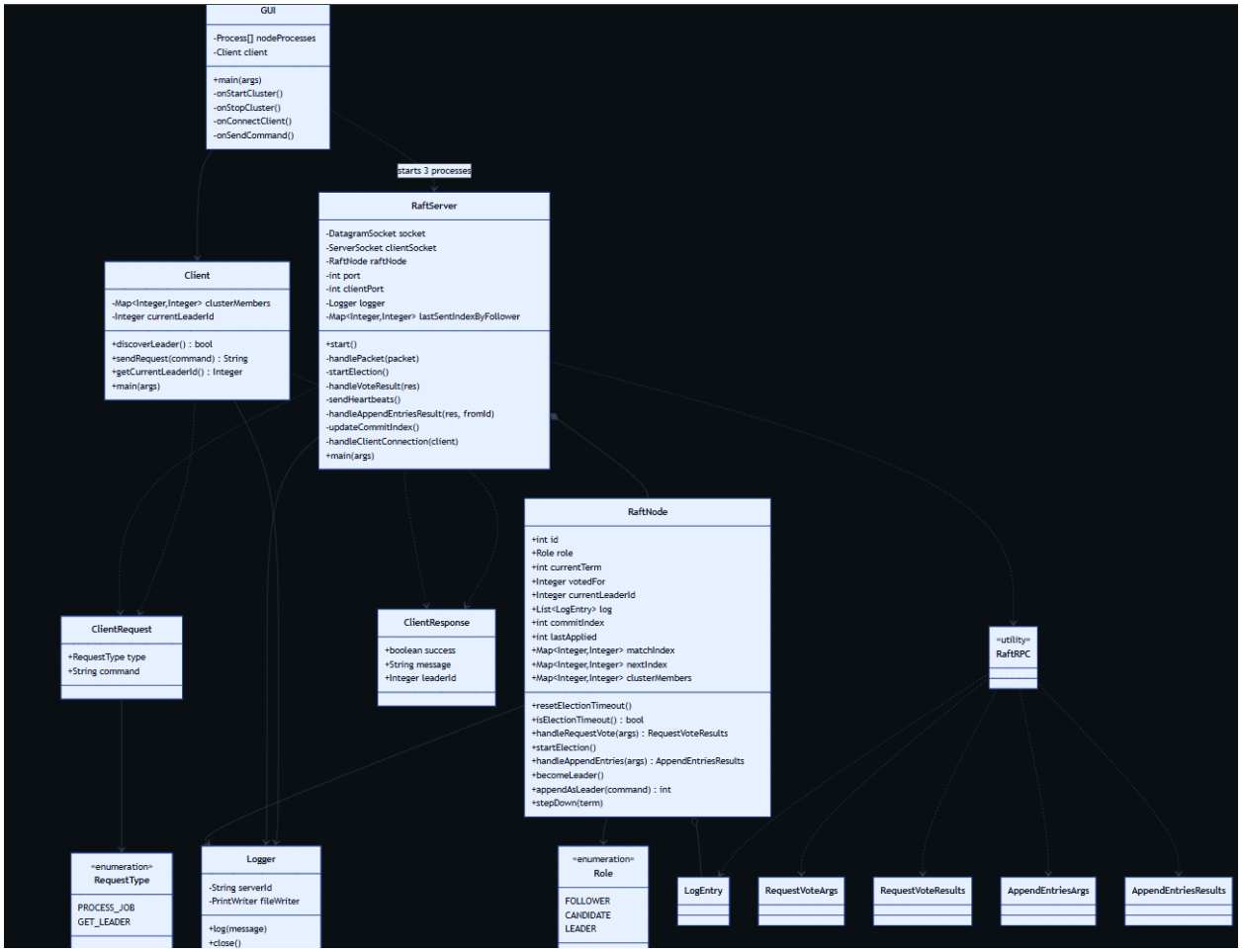


Fig 4: Sequence Client Failover

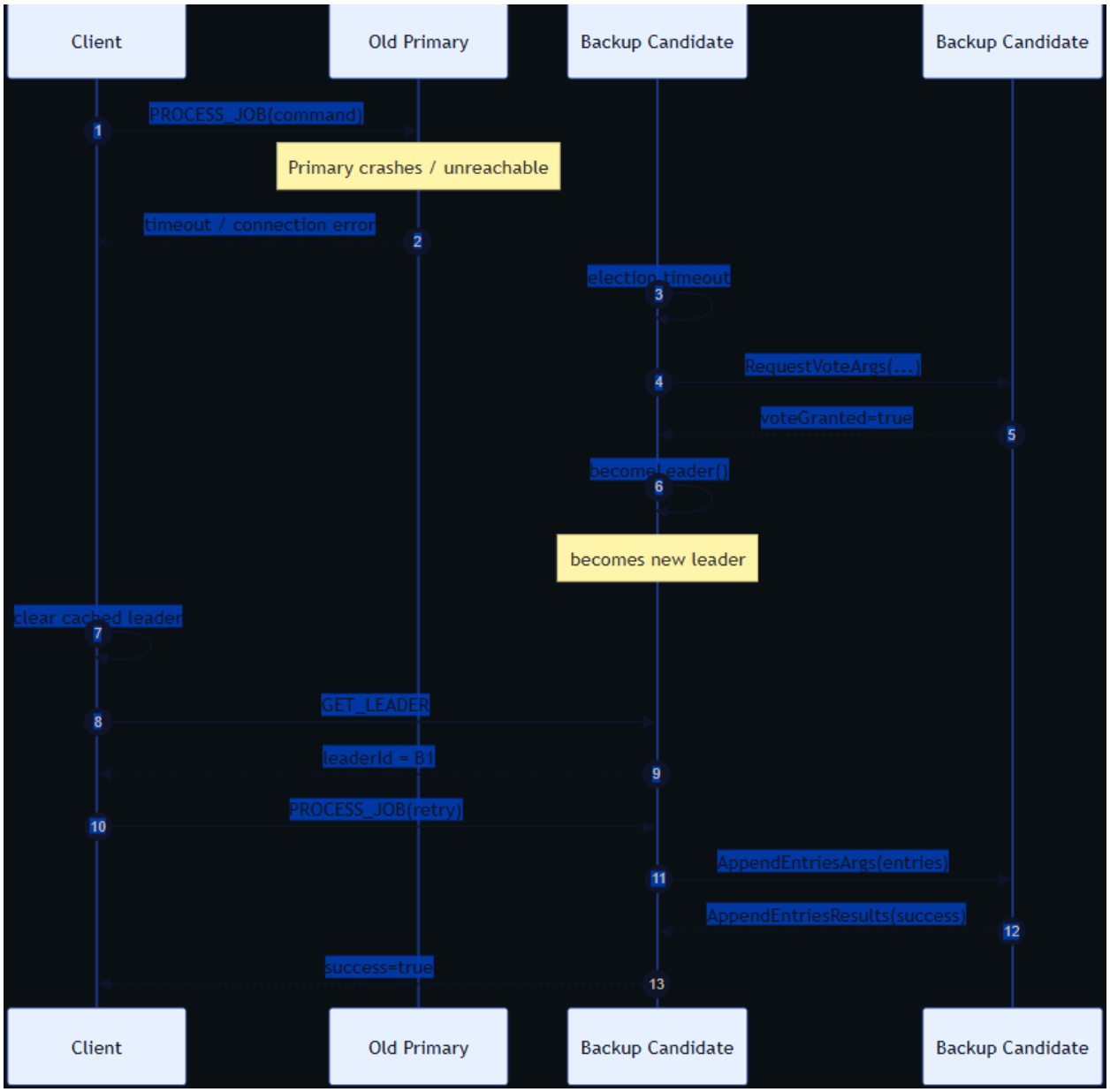

Client
Old Primary
Backup Candidate
Backup Candidate
PROCESS_JOB(command)
1
Primary crashes / unreachable
timeout / connection error
2
election timeout
3
RequestVoteArgs(...)
4
voteGranted=true
5
becomeLeader()
6
becomes new leader
clear cached leader
7
GET_LEADER
8
leaderId = B1
9
PROCESS_JOB(retry)
10
AppendEntriesArgs(entries)
11
AppendEntriesResults(success)
12
success=true
13
Client
Old Primary
Backup Candidate
Backup Candidate

Fig 5: Sequence Normal

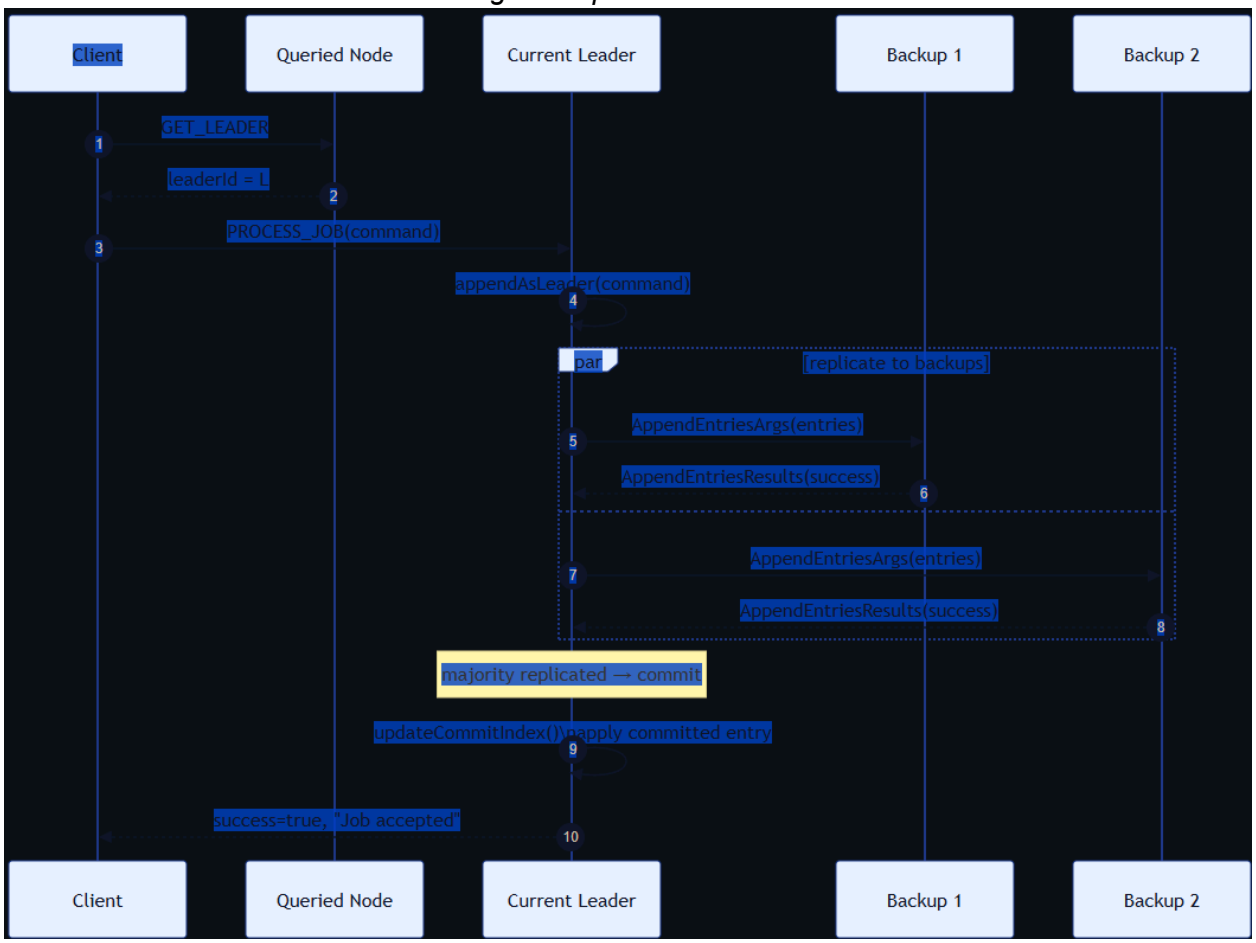


Fig 6: Sequence Startup and Election

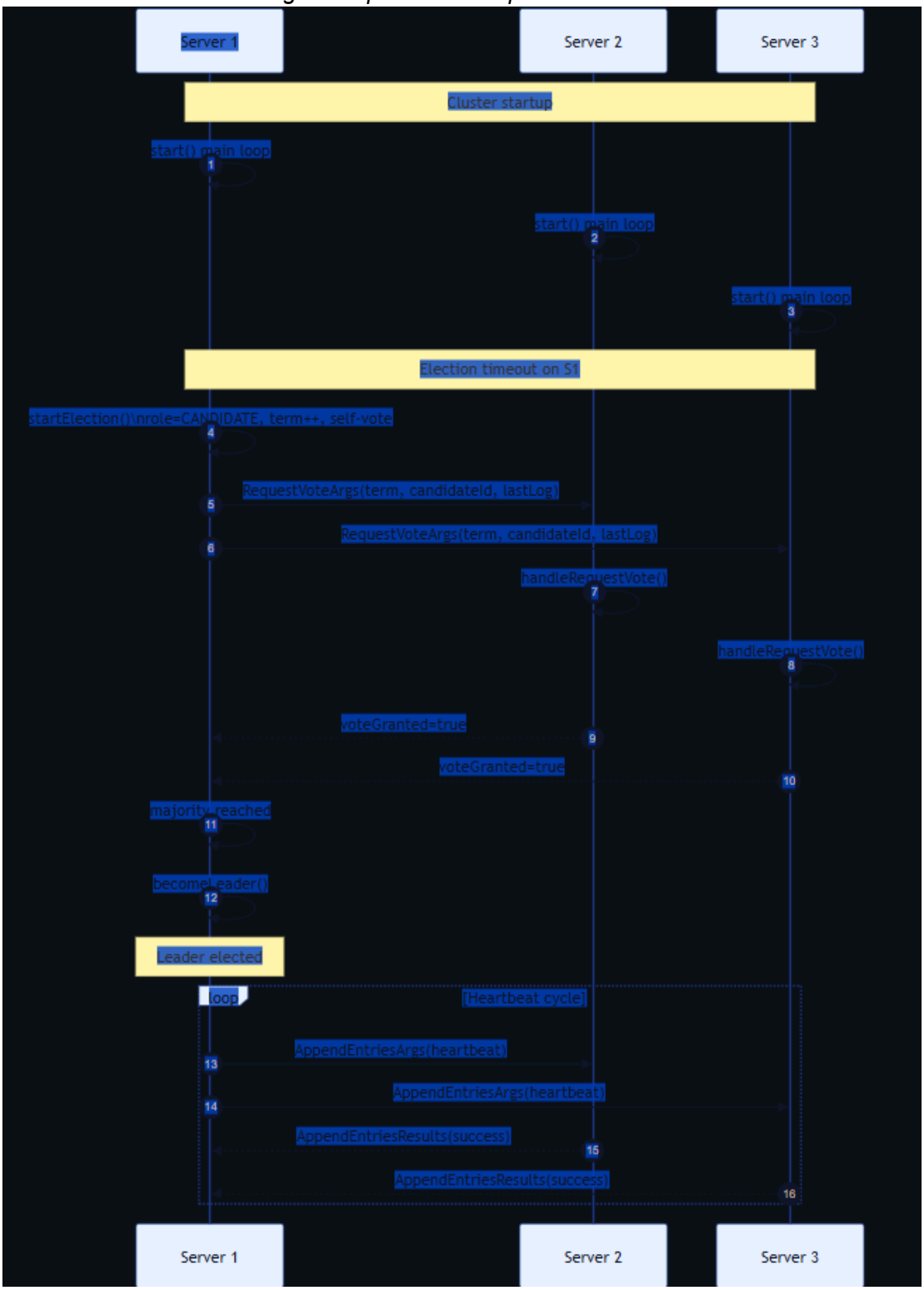
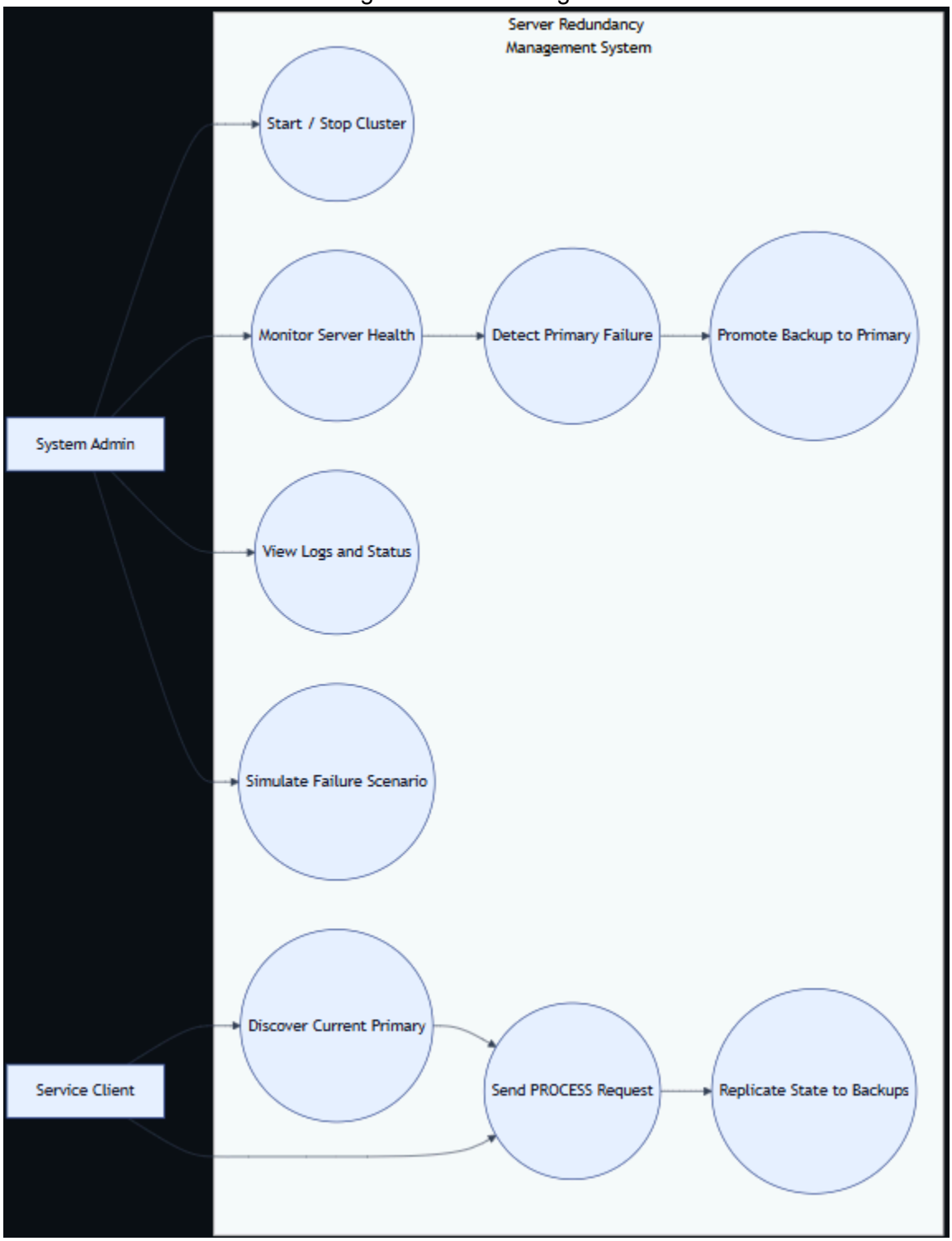


Fig 7: Use Case Diagram



*All these diagrams and a few more can be found on the Git and ZIP file.

VI. DISCUSSION

A.Trades Offs, Assumptions, and Design Choices

Trade Offs:

We are using both UDP and TCP and this is by design choice. I am quite familiar with web requests, and I did it this way because I know that we can UDP requests for more control. As we got further along with the project, and became more familiar with Raft systems, I discovered that this is not common and that using only TCP is preferred. Using TCP and UCP does(I think) make it easier to implement on a small scale but harder for large scale systems. This is something I want to change later.

The ports 9102-9104 and 8102-8104 are fixed. This was for simplicity and I have made some attempts to build a more dynamic system but failed. I will be spending some time during the reading break to look into how to implement this, to hopefully allow more than 3 nodes. I think using Java serialization was a good idea because it's quite simple.

GUI vs Headless. The entire project was headless at first. It was deprecated because it is harder to use. The GUI is a bloated mess of code, but intuitive. This is a maintenance vs ease of use tradeoff and does not really affect the algorithm or overall design.

Memory States. The Log and Raft states are not persistent so hard restarting all the nodes loses its memory. This is ok for a demo project but wont work well irl. We didn't spend much time on this as we knew it would be quite hard but this is a good place to improve.

Assumptions:

The big one is the cluster is 3 nodes. No other sizes. Another one is that we run on the same network and that the ports are pre-determined. We also aren't thinking about things like firewalls, ports that may already be used. A funny one I didn't think of was that we are using the system's time for the election clocks(whoops!). A lot of small things like this will make it hard to implement this on scale but are fine for a demo. Our client commands also actually don't do anything, there is no parsing or backend stuff going on(it's like a ping).

Design Choices:

Raft Algorithm: This was a great choice, very well documented lots of resources online and there is a great video series by MIT which I followed closely.

Using multiple threads. We have a thread for receiving the UDP and one for the TCP. I think the alternative to this is using one thread and a queue from what I've seen. This makes it simpler overall.

Worth switching to Maven? We saw that in the assignment PDF it mentions using Maven. Our current project does not use dependencies but I think that we would benefit from Junit testing and trying to find some external libs for helping with the logging(which is just printed to a large txt file). It will really help with the complexity going forward as we need to add more things, and I think it would be a good change.

B.Challenges

We faced many challenges during the project, but mainly the issues were with working as a team and not the practical content itself, such as the following.

Communication:

As always, when working with a team, there would be varying issues such as communication methodology. So as to resolve any issues in advance, we simply designated a singular means of communication, discord, so there would be no confusion moving forward on where we should message or talk if any issues or any other things arrived.

Scheduling:

We had issues with finding times to properly meet. We solved this by scheduling set days and set times to meet up on those days so we can regularly have progress updates, ask questions, and if any of us could not appear, we would be able to inform the others well in advance. This resolved the issue

Workload:

The final main challenge we had was how to divide the workload fairly so no one would do too much over another. We solved this by initially dividing the work, and if one task was too laborious, or much harder to complete then expected, we would have the group mate with the most free time go on to assist the group member struggling, leading to a fair and evenly divided workload for everyone.

Compatibility/Workflow:

Many people prefer to use different software and workloads that may not work together. We ended up using 3 start files to cover all these cases.

Future Improvements:

We will work towards not having these issues in the future by resolving them at the start, and setting up easy communication between members through discord as to prevent any new issues from arising.

VII. CONCLUSION

In conclusion, we successfully completed what we aimed for with the project objectives and requirements and delivered a satisfactory deliverable that we are proud to present. We learnt and experienced a lot during the project, dealt with many challenges, learnt how to work and overcome them as a team, and overall learnt many things about the development lifecycle.

VIII. REFERENCES

Our entire project is stored on https://github.com/Blake1332/COMP370-Project1-Group-3 The main algorithm was implemented here, along with the paper

- https://raft.github.io/
- Ongaro, Diego, and John Ousterhout. “In Search of an Understandable Consensus Algorithm (Extended Version).” *In Search of an Understandable Consensus Algorithm*, Stanford University, raft.github.io/raft.pdf.

MIT offers amazing and free CS content which was used during the implementation

“MIT Distributed Systems Series.” *YouTube*, www.youtube.com/watch?v=cQP8WApzIQQ&list=PLrw6a1wE39_tb2fErI4-WkMbsvGQk9_UB.

MIT 6.5840: Distributed Systems, MIT, https://pdos.csail.mit.edu/6.824/, Lab 3 (2025)