Lab Time: Friday 08:00 12:00

Group Number: 79

TTK4145 Preliminary Design Description

Group Members: [Names and emails to be added]

System Overview

We design a peer-to-peer (P2P) distributed elevator control system with three independent computers, each controlling one elevator across four floors. The system prioritizes **fault tolerance** and **request durability**. At most one computer fails at any time, ensuring at least two nodes remain operational and can preserve the global state of the system.

Core Design: Request Durability via RAM Replication

The fundamental challenge is ensuring no request is lost when nodes crash and network is unreliable. Our solution relies on network-level replication:

- **Immediate broadcast:** When a button is pressed (hall or cab call), the originating node broadcasts the request to all peers immediately.
- **Quorum confirmation:** The request is marked "confirmed" only after receiving acknowledgment from at least one other peer. This ensures the request exists on   92 nodes in RAM.
- **Replication:** Each node maintains an in-memory copy of all requests from all nodes. When a node crashes, surviving nodes retain all its requests.
- **Recovery:** On restart, a node requests full state sync from any active peer and resumes operation.

Note on Persistence: We have chosen to avoid disk writes for this project. Durability is achieved solely through real-time replication across the network, leveraging the fact that at least two nodes are always operational.

Programming Language: Go (Golang)

We chose **Go** as our implementation language. Go is designed with simplicity and concurrency as first-class concerns, which aligns perfectly with our needs. Since neither group member has prior experience with concurrency-focused languages, Go's straightforward syntax and built-in concurrency primitives (goroutines and channels) allow us to focus on the core distributed systems challenges rather than wrestling with complex language features or garbage collection intricacies. This lets us concentrate on the design decisions that matter: request replication, failure detection, and deterministic assignment.

Network Topology and Protocol Choice

We use a **fully connected mesh topology** where each node communicates directly with all others via **UDP**. We choose UDP over TCP for a critical reason: **TCP interprets high packet loss as connection failure and disconnects**, which would falsely trigger node failure detection in our system. UDP tolerates packet loss without disconnecting, allowing us to distinguish between transient network issues and actual node crashes.

The trade-off is that UDP provides no delivery guarantees, so we **design our own packet confirmation mechanism** using explicit ACK messages and application-level retry logic.

Handling Network Unreliability (Packet Loss)

Packet loss is expected and does **not** trigger system restart. Our custom reliability layer:

- **Retry with exponential backoff:** Failed broadcasts retry at 100ms, 200ms, 400ms intervals (max 3 5 attempts).
- **Explicit ACK messages:** Every ANNOUNCE message requires an ACK response. Missing ACKs trigger retries.
- **Distinguish transient from persistent:** After max retries, the peer is marked "unreachable" (likely crashed), not just "packet lost."
- **Heartbeat mechanism:** Periodic "alive" messages (500ms) detect actual node failures. Missing 3 consecutive heartbeats marks peer as down.

Hall Orders vs. Cab Orders: Different Handling

We distinguish between two request types:

- **Hall orders** (floor buttons outside elevators): Can be reassigned to any available elevator. If the assigned elevator's node crashes, other nodes reassign the order to a working elevator.
- **Cab orders** (floor buttons inside elevators): Cannot be reassigned because a passenger is in that specific elevator. If the node crashes, the cab order is preserved in RAM on surviving nodes and **resumed when the elevator comes back online**.

Deterministic Request Assignment

All nodes run the same assignment algorithm independently on the same request queue, ensuring they compute identical assignments without explicit coordination:

- Assign each **hall order** to the elevator with shortest estimated service time (considering current position, direction, and pending requests).
- **Cab orders** are always assigned to their originating elevator (no reassignment).
- Tiebreaker for hall orders: lowest node ID wins.

Button Light Contract

Button lights reflect the presence of a request in the system, but their visibility depends on the request type:

- **Hall Lights:** Turn **on** globally when a request is confirmed (exists on 92 nodes). They turn **off** globally when the request is completed.
- **Cab Lights:** Turn **on** only for the specific elevator handling the request. This ensures passengers only see the status of their own elevator's internal requests.
- **Persistence:** Light state survives node crashes because the underlying request is replicated in the global queue.

Handling Node Disconnection and Takeover

When a node becomes unreachable (network partition or crash):

- Other nodes detect via heartbeat timeout.
- **Hall orders** from the disconnected node are recomputed and reassigned to working elevators using the deterministic algorithm.
- **Cab orders** from the disconnected node are preserved in RAM but **not reassigned**. They wait for the elevator to come back online.

Error Handling and System Restart

System restart is triggered only by **persistent failures**, not transient packet loss:

- **Restart triggers:** Hardware failure (motor, sensors), unrecoverable logic error (exception), or all peers unreachable.
- **Restart action:** Stop all elevators, clear request queues, reinitialize network, sync state from surviving peers, resume.

Finite State Machine (FSM) for Elevator Control

We model each elevator using a small FSM for motion, while treating door behavior as **auxiliary data** rather than explicit states. This keeps the controller simple and avoids state explosion.

- **Core states:** Idle, MovingUp, MovingDown.
- **Auxiliary data (not states):** `currentFloor` , `doorOpen` , `doorTimer` , `targetDirection` , and the order sets (hall up/down and cab orders).

Transitions: Idle MovingUp/MovingDown when orders exist above/below. MovingUp/ MovingDown Idle when no further orders remain in the current direction. When arriving at a floor with pending orders, the controller opens the door by setting `doorOpen=true` , starting/resetting `doorTimer` , clearing served orders, then closing the door when the timer expires and continuing or re-evaluating direction.

Door Obstruction: If an obstruction is detected while the door is closing, the `doorTimer` is restarted, giving passengers additional time to clear the obstruction before the door attempts to close again.

Module Structure

We split the implementation into small modules with clear responsibilities to reduce complexity through **decoupling**:

- **Network:** UDP mesh, message formats, ACK/timeout/retry, and heartbeat send/receive.
- **State Management:** In-memory replicated request tables, confirmation state, and full-state sync on rejoin.
- **Elevator Controller:** Local motion FSM, door logic/timers/obstruction handling, and interface to the elevator driver.
- **Assignment Algorithm:** Deterministic hall-order assignment and tie-breaking; cab orders remain local.

- **Fault Detection:** Peer liveness tracking and trigger conditions for takeover vs. persistent failure.
- **Restart:** Centralized handling of unrecoverable failures (stop elevator, reinit network, resync state, resume).

Why This Design Works

No requests lost: Replication on 92 nodes guarantees durability. **Resilient to packet loss:** UDP avoids false disconnects; custom ACK/retry logic provides reliability. **No complex consensus:** Deterministic assignment avoids leader election or Paxos. **Respects passengers:** Cab orders are never reassigned and lights are localized to the specific elevator. **Language choice supports focus:** Go's simplicity and built-in concurrency let us concentrate on distributed systems challenges. **Handles all scenarios:** Crashes, disconnections, cab order preservation, and packet loss all addressed.