TTK4145 - Real-time Programming Report

[REDACTED]

Januar 2025

Contents

1 Introduction

This project presents the design and implementation of a fault tolerant, scalable, and efficient real time elevator system. The goal is to handle concurrent requests while ensuring synchronized operation across multiple elevators and maintaining robustness against failures. To achieve this, the system leverages distributed computing, multi threading, and real time event handling.

A major challenge in elevator control is managing slow I/O signals without compromising responsiveness. By using multi threaded processes, the system efficiently processes inputs and outputs in parallel, preventing delays that could hinder real time performance. Shared memory and inter process communication ensure that the system reacts promptly to user input, request updates, and emergency conditions.

To eliminate single points of failure, the architecture is designed as a distributed system where core functions are handled independently. Database nodes ensure data persistence and synchronization, manager nodes handle request assignment and scheduling, while elevator nodes execute assigned tasks and maintain real time status updates. Inspired by the Data Distribution Service (DDS) model, the system maintains synchronized operations across all components, dynamically redistributing tasks to prevent service disruptions.

By integrating real time state machines, cooperative scheduling, and robust error handling mechanisms, this project delivers a reliable and scalable solution for modern elevator control, ensuring efficient request handling, fault tolerance, and smooth operation under various conditions.

Throughout the project, AI tools like ChatGPT and DeepSeek helped with coding, debugging, and understanding technical documents. AI also improved the reports grammar and clarity, making it more readable and professional.

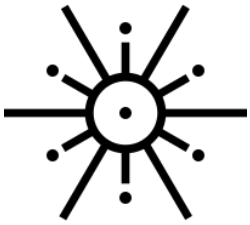
2 Software Architecture

2.1 Rust

Rust was chosen for this project due to its strong focus on thread safety, memory management, and performance. Its preemptive threading model, which focuses on task execution order determined by the operating system, is effective but not ideal for scalable real-time solutions. For this, cooperative scheduling is preferred, as it allows better control over resource allocation and task execution. Rust excels in preventing common concurrency issues, such as race conditions and undefined behavior, making it a robust choice for multi-threaded systems.

However, Rust's syntax can be verbose, making it less intuitive for developers new to the language. Additionally, while Rust is highly capable, it lacks native support for DDS (Data Distribution Service), which complicates implementing certain real-time and distributed functionalities without external libraries or workarounds.

2.2 Tokio



To transform Rust into a cooperative real-time system, we rely on the Tokio framework. Tokio provides an asynchronous runtime that enables us to write concurrent programs using lightweight *"tasks"*, which are more efficient than traditional threads. These tasks are a hybrid solution between full OS threads and fibers, designed to minimize overhead while maintaining flexibility. For IO-bound systems like our elevator, where operations must wait for external signals or inputs, Tokio's asynchronous model ensures that the CPU remains active and efficiently utilized.

Tokio is widely adopted in industry, powering large-scale systems such as Amazon's infrastructure and Discord's servers. This adoption ensures strong community support, robust libraries, and practical solutions for real-world issues. Additionally, Tokio is real-time agnostic, meaning it can be adapted for various scenarios without sacrificing its cooperative nature.

One critical consideration is that cooperative systems require the developer to yield control of tasks explicitly. Failing to do so can lead to starvation or deadlocks, where some tasks monopolize resources. However, Rust's thread safety and Tokio's task scheduling mitigate most issues if designed correctly. While Tokio is highly effective for soft real-time systems, its cooperative nature may not guarantee 100% real-time performance. Nevertheless, it achieves good enough efficiency for most applications that deal with inputs and outputs as bottleneck, one of them being our elevator system.

For resource-constrained environments like microcontrollers, Tokio's resource requirements may be excessive. However, for devices such as Raspberry Pi, Rock Pi, or standard computers, Tokio is an ideal choice, combining performance with scalability.

2.3 Zenoh

Zenoh was selected to simplify data communication and synchronization across distributed nodes. It provides an abstraction over low-level networking, unifying data in motion, data at rest, and computations. By supporting a peer-to-peer model, Zenoh eliminates reliance on central hubs and adapts seamlessly to various network topologies, including local and wide-area networks. Its flexibility enables efficient communication across devices, from microcontrollers to data centers, making it ideal for scalable distributed systems.[1]

While Zenoh includes fault tolerance and scalable communication, it does not natively handle data synchronization across nodes. To address this, a custom leader election algorithm was implemented in this project, enabling nodes to dynamically select a leader responsible for managing data distribution. This manual synchronization ensures that the system operates reliably, complementing Zenoh's robust networking capabilities and addressing specific requirements for real-time synchronization.[2][3]

Despite not matching the raw performance of protocols like DDS or UDP/TCP in niche scenarios, Zenoh excels in scalability, manageability, and minimal resource usage. It enables efficient data flow with low overhead and supports diverse network protocols. By integrating Zenoh's features with custom synchronization logic, the project achieves a balance between flexibility and performance, making it well suited for real-time applications such as our elevator system.[4]

2.4 Advantages and Disadvantages

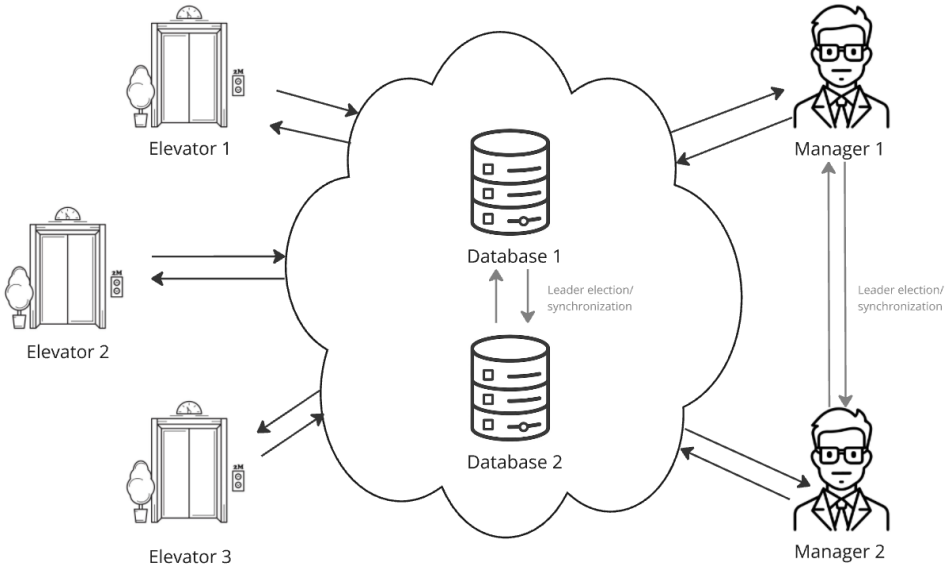
Advantages:

- *Fault Tolerance*: Distributed nodes handle failures gracefully, ensuring system continuity even if one node fails.
- *Scalability*: The cooperative model of Tokio and the distributed nature of Zenoh make it easier to expand the system to more nodes or features.
- *Developer Productivity*: Tokio and Zenoh abstract complex details of multithreading and networking, reducing the risk of bugs and speeding up development.
- *Real-Time Performance*: Provides good performance for soft real-time systems, sufficient for most applications like our elevator system.

Disadvantages:

- *Cooperative Scheduling Overhead*: Explicit task yielding is required, and failing to do so can lead to deadlocks or unresponsive behavior.
- *Higher Resource Usage*: Compared to raw UDP/TCP, Tokio and Zenoh consume slightly more memory and CPU, making them less suitable for constrained environments.
- *Verbose Syntax*: Rust's syntax, combined with the additional layers of abstraction from Zenoh and Tokio, can feel cumbersome for new developers like ourselves.

2.5 Elevator System Overview



The elevator system is designed using concepts borrowed from the Distributed Data Service (DDS) model, where data is centralized and redundantly backed up across multiple nodes. This ensures fault tolerance, as any database node can temporarily assume leadership if another fails. Once the failed node recovers, it synchronizes with the latest data from the central database and resumes normal operation seamlessly. This architecture provides resilience and minimizes downtime in the event of failures.

Action nodes, responsible for implementing state machines and controlling elevators, use the central databases to exchange data and coordinate operations. These nodes have fault tolerance mechanisms that allow others to take over their roles during failures. When a failed node reenters the network, it retrieves the latest updates from the database and uses internal algorithms to determine its role and responsibilities. This dynamic recalibration ensures that the system remains operational and efficient at all times.

Separating the elevator and manager nodes enhances scalability, as a few manager nodes can handle multiple elevators in large systems. This division simplifies logic and data flow, enabling parallel processing and more efficient operation. While we could have integrated the manager node into the elevator node, there is a clear distinction between the two, making them well suited to be separate processes. The primary advantage is that each elevator operates independently, focusing solely on its tasks without needing direct synchronization with the entire system. All incoming and outgoing data is assumed to be synchronized, allowing the database and manager nodes to handle synchronization, backup, and network wide data distribution with minimal overhead.

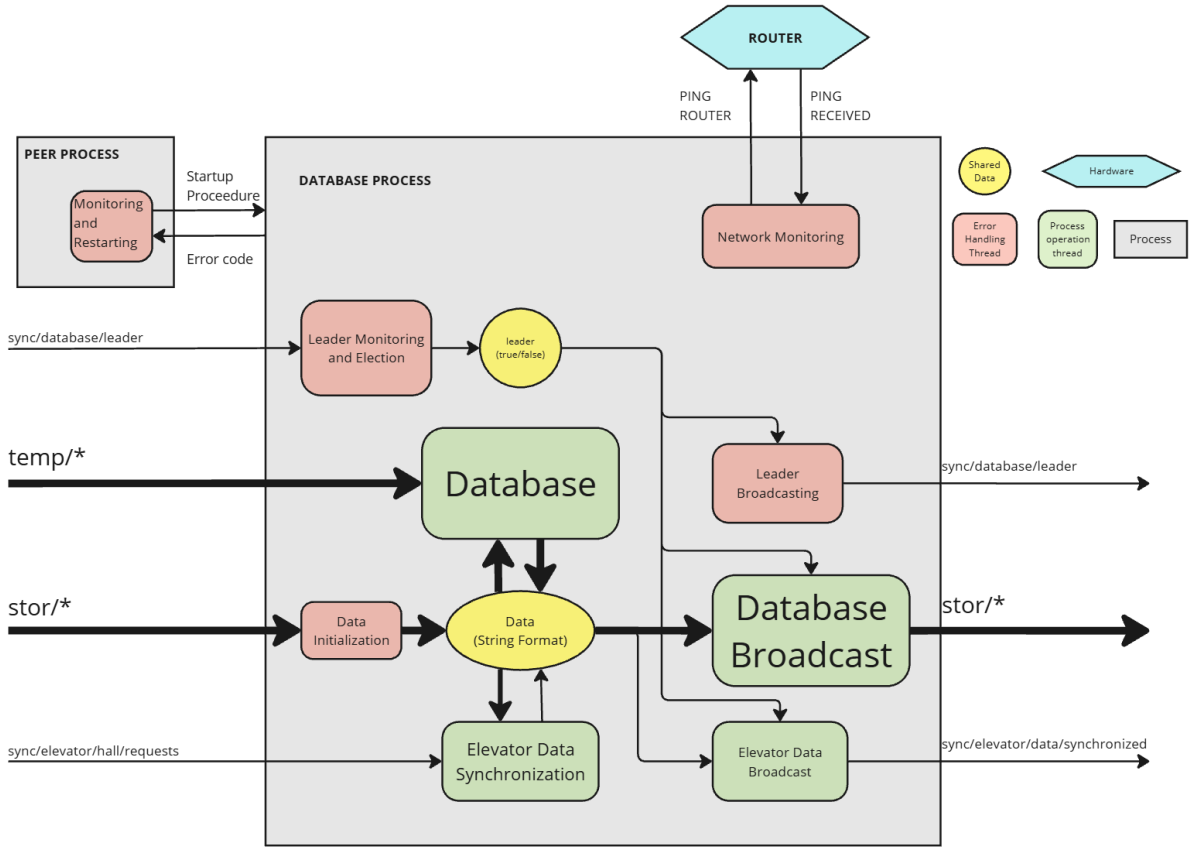
This separation simplifies the logic, making nodes more manageable. Elevators can focus on executing their tasks efficiently, while the database ensures synchronization and data persistence throughout the distributed network. The manager node, in turn, oversees the entire system, ensuring synchronization while remaining aware of all operations. This clear separation of roles enhances maintainability, scalability, and system tuning, allowing for easier modifications and optimizations over time.

The manager node is responsible for handling all incoming requests and determining the most suitable elevator for a floor request using cost functions. It processes data from the central database, makes scheduling decisions, and updates the database with optimized assignments. The updated instructions are then distributed to all elevator nodes for execution. If a manager node fails, a backup manager takes over, ensuring continuity until the original node rejoins.

Shared features, such as peer process management, network connectivity monitoring, and synchronization mechanisms, are implemented across nodes to maintain system robustness and fault tolerance.

3 Database Node Design

3.1 Overview



The database process in this project is designed to ensure fault tolerance, scalability, and efficient real-time data management through a peer-to-peer architecture. By distributing responsibilities across nodes, the system eliminates single points of failure. Synchronization is achieved via a combination of Zenoh's built-in distributed data synchronization and custom leader election algorithms for deciding who is going to distribute data. On initialization, nodes listen for stored data from other nodes to ensure they begin operations in sync with the network, enabling seamless integration into ongoing processes. The combination of Rust, Tokio, and Zenoh forms the foundation of this architecture, delivering thread safety, asynchronous processing, and distributed synchronization in a robust platform for real-time systems.

The database process ensures fault tolerance, scalability, and real-time data management using a peer-to-peer architecture. Nodes eliminate single points of failure by distributing responsibilities and synchronize. This synchronization is done via Zenoh's inbuilt network configuration file we modify for our needs. In addition we use custom leader election algorithms for deciding who is going to distribute data for seamless transportation of data without clashes. On initialization, nodes sync with others to start with the latest data.

3.2 Network Monitoring and Recovery

Each node continuously monitors its network connection using ICMP pings to detect disconnections from the router. If a node loses connectivity, it terminates itself to prevent stale or inconsistent data from propagating through the system. The process pair mechanism actively monitors the state of the network and restarts the database process after a determined amount of time. This design minimizes downtime and ensures smooth reintegration of nodes into the network.

The peer-to-peer architecture removes dependency on a central server, enhancing the systems resilience to network disruptions. Nodes communicate and synchronize directly with one another, ensuring data consistency and robust recovery in challenging conditions.

3.3 Leader Election and Synchronization

Fault tolerance is primarily achieved through leader election and peer-to-peer communication. At any given time, the leader node is responsible for publishing data, while other nodes listen passively. In the event of a leader failure or disconnection, a new leader is dynamically elected based on predefined node priorities, lowest node ID wins the election. This ensures seamless data transmission and continuity of operations.

The process pair mechanism enhances reliability by monitoring node health and restarting the database process if a failure occurs. When a node rejoins the network, it retrieves the latest data state and reenters operations without disruption. By combining decentralized leadership roles with efficient recovery mechanisms, the system achieves high availability and fault tolerance.

3.4 Data Storage and Transmission

At startup, each node listens for stored data from other nodes, ensuring it initializes with the most recent data state. This step guarantees that all nodes are synchronized from the go. Updates to shared data are managed by the leader node, which broadcasts changes to all nodes in the network. This approach ensures consistency across the system, even during leadership transitions.

The custom leader election algorithm integrates with Zenoh's synchronization features to efficiently manage data transmission and consistency. By leveraging this combination, the system guarantees accurate and reliable data updates while adapting to changes in network conditions or node states. The architecture's focus on consistency and robust transmission ensures reliable performance even under dynamic conditions.

3.5 Elevator Data Synchronization

The overall system relies on the manager being synchronized with all elevators. If the manager has up to date data from all the elevators, it can accurately distribute requests to each elevator. Additionally, a byproduct of synchronization is that all elevators share the same dataset, ensuring persistent information across all nodes. This means each elevator can understand what other elevators are doing without direct interaction. To achieve this, we implement the Elevator Data Synchronization functionality within the database node.

This synchronization algorithm collects new data from each elevator, which is relevant to both the manager node and other elevators. This data includes elevator states and hall requests. The data is then stored in the shared data storage. If the database node is the leader, it is already synchronized, and its role is simply to distribute this data across the network. Furthermore, an optimization algorithm monitors changes and only transmits data when modifications occur. This prevents redundant transmissions, such as when a hall button is repeatedly pressed. Once the data is synchronized and confirmed, there is no need for rebroadcasting, reducing network bandwidth usage and ensuring that the manager is called upon only when necessary.

A crucial aspect of the elevator data synchronization process is that all messages are formatted in JSON. This JSON format specifies whether to ADD or REMOVE specific hall requests from the database. Due to the synchronized nature of the data, only one elevator at a time can handle a specific floor request,

reducing conflicts where multiple elevators attempt to remove the same hall request. If such a conflict does arise, where the request has already been removed, the second removal request is simply ignored to handle this edge case.

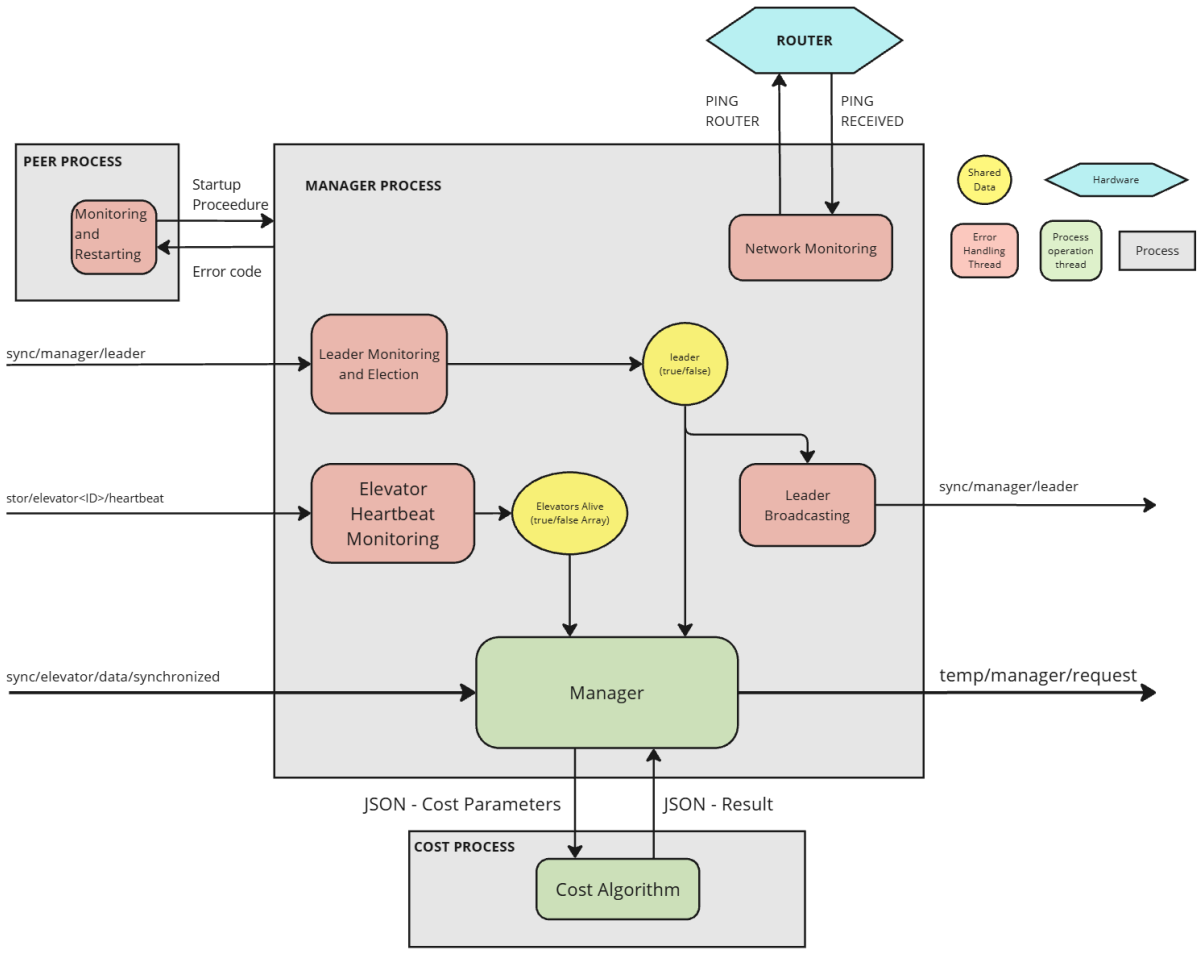
Conflicts typically occur when the manager changes its decision. In such cases, the second elevator will detect the change where first elevator already handled the request through the synchronized data. Instead of continuing toward a now invalid request, the elevator will transition to an idle state at the nearest floor, ensuring optimal request handling and avoiding unnecessary movements. This approach enhances efficiency and ensures a seamless elevator coordination system.

3.6 Drawbacks

A potential edge case occurs with three database nodes or more when the leader disconnects, a new leader is elected, and the old leader reconnects as the new leader dies. In this scenario, the old leader might dominate the election before receiving updated data, causing data inconsistency and desynchronization. To mitigate this, the initialization period is extended, allowing the last active node to propagate updated data before the old leader resumes leadership, ensuring consistency. This makes system momentarily slow on failures until the process is all up to speed and synchronized again.

4 Manager Node Design

4.1 Overview



The manager node is responsible for task allocation, determining which elevator is best suited for each request by calculating cost functions. It listens to synchronized data from elevator nodes, tracks their status, and distributes tasks accordingly. In case of failure, a backup manager seamlessly takes over to ensure uninterrupted operation.

Separating the elevator and manager nodes enhances scalability, as a few manager nodes can handle multiple elevators in large systems. This division simplifies logic and data flow, enabling parallel processing and more efficient operation.

4.2 Fault Tolerance

The manager node borrows fault tolerance concepts from the database node. Peer process monitoring, network supervision, and synchronization are implemented similarly, with only variable and topic name adjustments.

Additionally, a custom monitoring thread is implemented within the process. This thread tracks the status of all elevators. If an elevator disconnects or experiences a failure, its heartbeat signal stops. After a few seconds without a response, the monitoring thread assumes the elevator is offline and shares this information with the manager thread. Once the elevator reconnects, its status is updated and shared accordingly.

4.3 Manager

The manager node gathers synchronized data from the database regarding each elevator. It only receives updates when data changes, such as an elevator changing state, receiving new cab calls, moving to a different floor, or new hall requests being registered. The goal is to aggregate all relevant data efficiently.

Before proceeding with any calculations, the manager node checks if it is the current leader. If it is not the leader, the process halts to avoid unnecessary resource consumption. If the manager node is the leader, it reformats and cleans up the received data. It also checks if any elevators have gone offline and removes them from further calculations.

Once the data is formatted, it is converted into a JSON structure and sent to a separate process responsible for performing complex cost function analysis. This process determines which available elevator should handle each request. The result is returned in another JSON format, which the manager node then forwards to the rest of the system. The database ensures the information is backed up and synchronized before distributing the instructions to the elevator nodes.

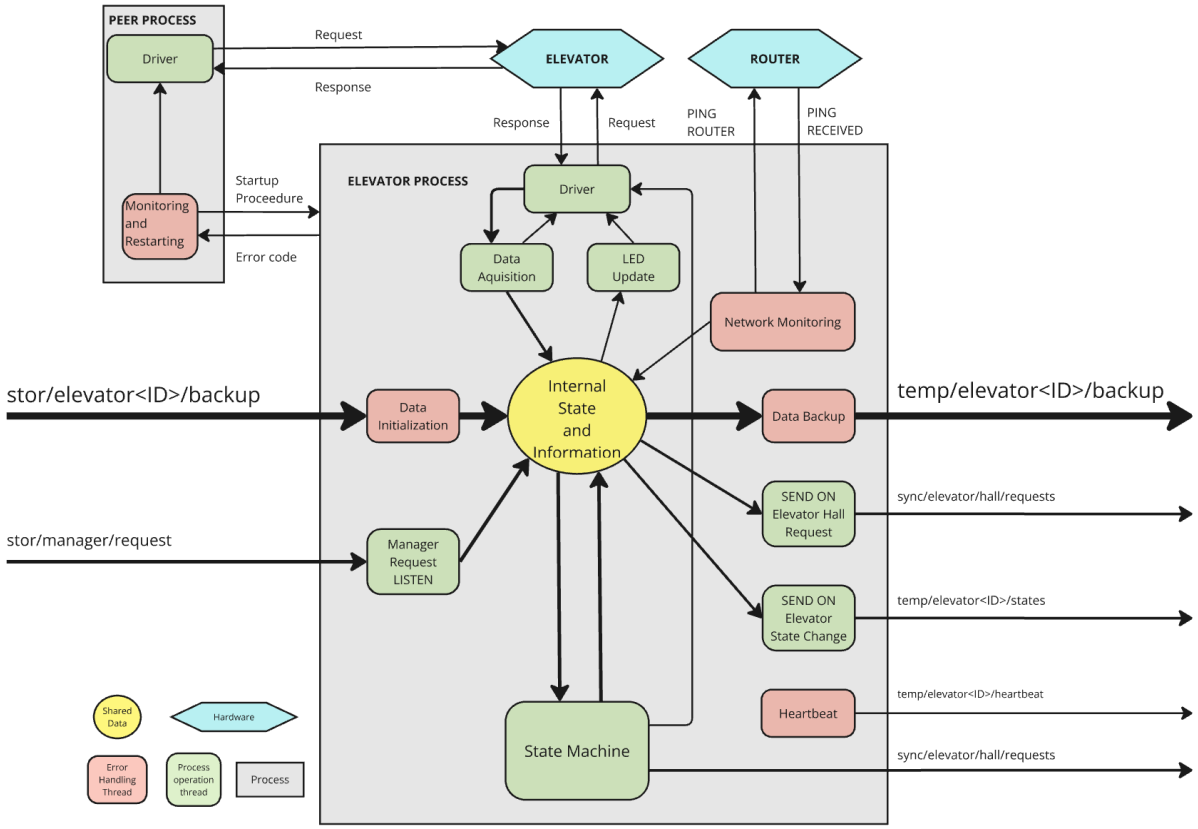
4.4 Advantage

Unlike the database node, which must track and synchronize all system data, the manager node does not require long term data storage. This means that the manager does not suffer from the same disadvantages as the database node. If the leading database crashes and requires reinitialization, the system experiences downtime while data is restored and synchronized.

With the manager node, there is no need for such downtime. Since all critical data is stored within the database, the manager node can immediately resume operations after a failure. A backup manager can take over leadership instantly, using the latest synchronized data, ensuring continuous system operation without delays.

5 Elevator Node Design

5.1 Overview



Elevator nodes manage the hardware of the system, specifically the elevator itself. They are responsible for gathering real-time data on the elevators status while simultaneously controlling its movements. These nodes communicate with the Manager Node, sending hall requests and receiving feedback on task assignments. Additionally, they transmit heartbeat signals to the Manager Node to confirm their active status.

To ensure fault tolerance, all collected data is externally backed up in the distributed database. If an elevator node shuts down, it can retrieve its last known state upon restart and resume operations seamlessly. The core logic driving the elevators behavior is a state machine, which processes incoming data to determine the next course of action efficiently.

Given the complexity of managing multiple processes and shared resources, significant effort has been invested in optimizing resource utilization. Clever concurrency management strategies minimize locked resource usage, reducing downtime and improving overall system responsiveness.

5.2 Network Monitoring

The network monitoring thread functions similarly to other network monitoring threads in different nodes. However, instead of killing the entire process upon network disconnection, it simply updates a shared boolean variable to indicate the network status. This approach is particularly useful for handling global data, such as the hall request queue sent by the manager and database.

In the event of a network disconnection, the system is notified that the elevator is offline. This allows the elevator to focus solely on processing internal requests without relying on external data. Once reconnected, it can resume normal operations with synchronized global data.

5.3 Hardware interaction

Driver Library
Driver is responsible for direct hardware communication, providing a low level but easy to use abstraction for message passing via TCP sockets to the elevator hardware port. This ensures every message sent to the hardware receives a response. However, hardware communication is significantly slower than CPU processing, requiring careful management to prevent excessive TCP socket usage. The driver is only accessed when necessary to optimize performance.

LED Update Threads
LED update threads listen to the internal queue and process state, storing relevant information locally to minimize shared resource access. These threads update elevator lights only when queue data changes, preventing redundant commands from being sent to lights that are already active. This optimization significantly reduces driver usage, allowing for faster interaction with the elevator for status updates.

Additionally, the LED update thread monitors the global hall request queue. This information is received from the manager node, which maintains synchronized data. The data is sent to the elevator and stored by the Manager Request LISTEN thread. The LED update thread reads this data to determine which hall requests are currently active across the entire network and adds them to the LED update queue.

If the elevator disconnects from the network, the LED update thread receives a notification through the shared resource managed by the network monitoring thread. In the event of a disconnection, all global hall request LEDs will be turned off, leaving only local hall request LEDs active. This prevents user confusion by ensuring that hall request LEDs do not remain illuminated when the elevator is no longer part of the network. Otherwise, users might assume an elevator is still responding to requests when it is actually offline and doesn't know about global hall request status.

Once the elevator reconnects to the network, the LED update thread resumes reading updated manager request data and determines which global hall LEDs need to be turned on again. This ensures that every elevator in the network remains fully synchronized. When disconnected, an elevator will only display local hall requests that it is actively handling.

Data Acquisition Threads
Data acquisition threads consume the majority of driver time. These threads request real time data from the elevator, such as cab and hall requests, and most critically, the current floor. The current floor data is polled over 20 times per second to ensure the state machine can react precisely and stop at the correct floor without delay.

State Machine Direct Driver Access
For improved responsiveness, the state machine has direct access to the driver. Whenever the state machine transitions, it must update the motor direction. Instead of using shared resources, which could introduce latency, the state machine directly manipulates motor controls. To minimize driver workload, the system only sends motor commands when the motor state or direction changes, avoiding unnecessary commands when no state change occurs.

Safety Measures for Process Failures
In the event of an unexpected process crash while the elevator is moving, a peer process monitors system health. This backup process operates a simplified, synchronous version of the driver without multithreading. Upon detecting a failure, it immediately sends a STOP signal to halt the elevator before restarting the main process. This ensures passenger safety and prevents unintended movement while the system recovers.

5.4 Request Queue Management

For cab queue requests, management is straightforward as the state machine directly handles them. The system always prioritizes passengers inside the elevator, ensuring that even if the elevator disconnects from the network, it can still process cab calls independently.

For hall requests, the process differs slightly. These requests are handled by the SEND ON Elevator Hall Request thread, which checks what type of hall butto was requested UP or DOWN, then it checks witch floor, all this information then gets packaged into a JSON format to decide what button was pressed and witch floor, the JSON dictaed witch floor on what hall button type queue will be addad. This data is then sent to the database. The database recieves it and reads the JSON, the corresponding add request will be activated and the specific floor will be added to the hall request queue for UP/DOWN deoending if it is already on the queue or not. From here the database sends the request to manager node if data has changed or not.

When the Manager Node sends a hall request assignment back to an elevator, a dedicated listener thread processes this response and updates the elevators hall queue accordingly. In adition all the data manager sent to elevator will be saved internaly in the shared resources, this way other threads like LED update therads can use manager information to synch their global hall LED status with what manager sees. Ensuring all Hall LEDs are in sync with the rest of the elevators.

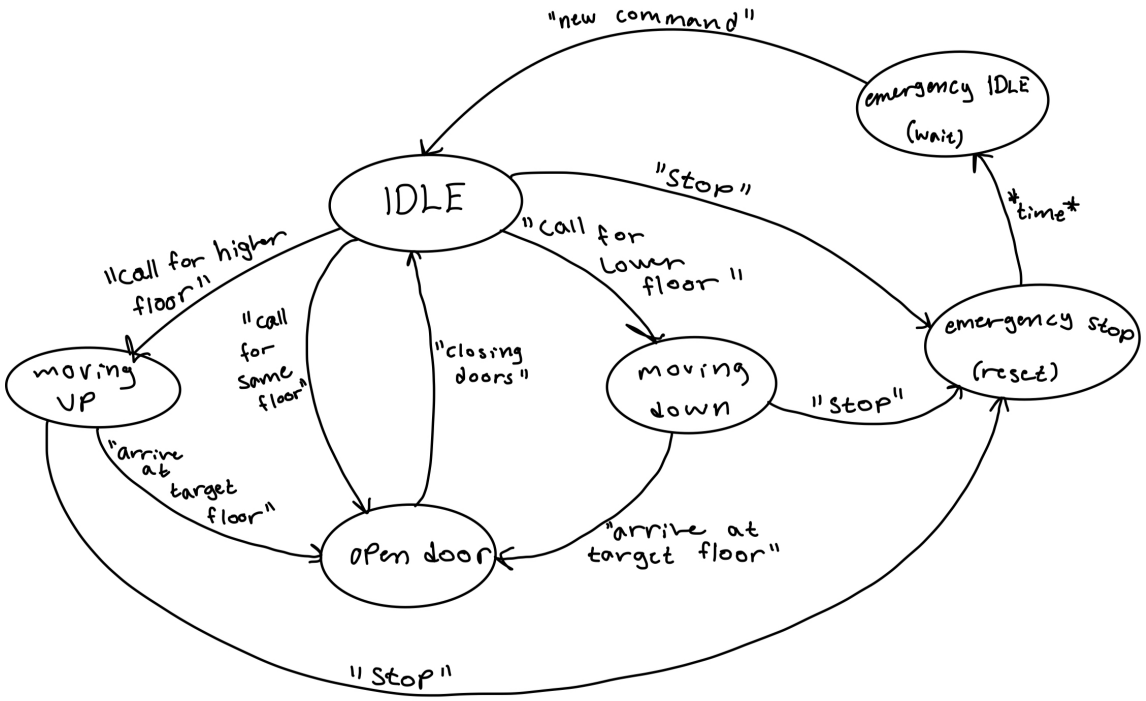
This double checking mechanism provides a key fault tolerance feature. If an elevator disconnects from the network, its hall requests will not reach the Manager Node, and consequently, the elevator will never receive new hall requests while offline. Instead, it will continue fulfilling only its pending hall requests and cab calls until it reconnects. This ensures that hall requests are only handled by active elevators, maintaining system reliability and avoiding orphaned requests.

5.5 Data Backup

There is always a possibility that the elevator process crashes while performing tasks. To ensure data integrity and continuity, the system consistently backs up elevator data to the distributed database.

In the event of a crash, before transitioning into state machine mode, the process first initializes all necessary data and checks if any previous backup exists. If a backup is found, the system restores the last saved state and continues operation from where it left off. This ensures that the elevator resumes seamlessly, preventing the loss of requests and maintaining operational efficiency.

5.6 State Machine



The elevator system utilizes a custom built state machine designed to efficiently handle various scenarios. It consists of 6 states total, including 2 emergency stop states and 4 normal operation states. The state machine reads and writes all its data from/to shared resources, including sensor inputs and network commands, such as manager requests. With the only exceptions being direct access to elevator driver for motor control and direct sending of hall request removal commands to synchronize the network.

When a user presses a hall button, the request is added and synchronized across the entire system. However, this does not guarantee that the manager node assigns the request to the specific elevator in question. Instead, the manager redistributes requests dynamically to optimize efficiency. On the other hand, if an elevator receives a request directly from the manager, its queue is updated accordingly. As a result, the elevator state machine must efficiently process requests in real time.

Once the elevator completes a request, it removes it from the queue. If the request was a hall request, the system must also synchronize this information across the network to ensure other elevators recognize that the request has been fulfilled. This triggers the necessary updates, such as turning off the corresponding LED indicators and adjusting the state machine logic accordingly.

This synchronization is achieved by sending a removal command for the hall request over the network. The message format used for removal is identical to the one used for adding hall requests, handled by the SEND ON Elevator Hall Request thread (ie JSON format). The database processes the removal globally and ensures all other elevators and manager nodes receive the updated information and all data is synchronized. By maintaining real time synchronization, the system remains scalable, responsive, and easy to manage.

Idle State
The Idle state is the default state entered at startup unless a previous state is restored. Upon entering this state, the elevator first checks the last movement direction and ensures the motor is in the correct state. If the elevator was previously moving, it does not immediately stop the motor but instead evaluates pending requests before determining the next action. The motor state is only changed when necessary to prevent unnecessary stops and delays.

When a cab or hall request is received, the system first checks for emergency stop conditions. If an emergency stop is triggered, the elevator immediately transitions to the emergency stop state. If no emergency occurs, the system evaluates whether the request corresponds to the current floor. If so, the elevator transitions directly into the door open state. Otherwise, the system determines whether it should continue in the previous movement direction or adjust based on newly assigned requests.

If multiple requests exist, the system calculates the optimal movement path to handle them efficiently. For example, if an elevator at floor 1 receives a down request from floor 3 and an up request from floor 4, it will first proceed to floor 4 before descending to floor 3, ensuring minimal movement and maximum efficiency.

Movement States (Up and Down)
The movement state is responsible for handling active travel between floors. When an elevator is assigned a request, it transitions into the appropriate movement state assigned from the Idle state.

If an emergency stop is pressed at any time, the elevator immediately transitions to the emergency stop state. Otherwise, upon reaching a floor, the elevator momentarily enters the Idle state but retains memory of its movement direction. Idle state then evaluates whether it should stop at the current floor or continue moving immediately. If there are no relevant requests for that floor, the elevator transitions back into the movement state without stopping, preventing unnecessary halts.

This system also enables real time hall request reallocation. If an elevator loses its hall requests due to manager reassignment while in route, it will not continue toward an outdated request. Instead, when reaching the next floor, it verifies its updated request queue and remains idle if no new requests exist, preventing redundant movement and ensuring synchronized system operation.

To avoid inefficiencies, the elevator ignores spammed floor requests immediately after departure. This prevents unnecessary reversals and maintains orderly queue management, ensuring smooth and logical elevator operation at all times.

Door Open State
The door open state activates when the elevator reaches its designated stop. The doors remain open for three seconds, after which the system checks for obstructions. If any are detected, the doors remain open until the obstruction is cleared. Once cleared, the door closes, then the system transitions back to Idle state and determines the next movement direction.

Emergency Stop States
If the emergency stop button is pressed, the elevator immediately halts, clears all pending requests, and enters an emergency idle state. New requests must be received before normal operation resumes.

However, a limitation exists. If the elevator stops between floors, it cannot automatically return to the previous floor due to hardware sensor constraints. Instead, the system remains in an emergency idle state until a new request is received, at which point it resumes normal operation. This approach prioritizes passenger safety and maintains a simple yet effective recovery mechanism.

5.7 Fault Tolerance

The elevator system is designed with multiple layers of fault tolerance to ensure robust and reliable operation:

- **Heartbeat Mechanism:** The elevator continuously sends heartbeat signals to the network, allowing the Manager Node to detect its presence. If an elevator goes offline, the Manager Node can reallocate hall requests accordingly.
- **Hardware Failure Handling:** If the elevator hardware loses power, the elevator process detects this and terminates itself. Upon hardware recovery, the process automatically reinitializes and restores backed up data.
- **Network Disconnection Handling:** If the network is lost, the elevator node attempts to reconnect. During disconnection, it continues handling cab calls and completes ongoing hall requests, but it will not accept new hall requests. Once reconnected, it resumes normal operation.
- **Process Crash Recovery:** If the process crashes, it automatically restarts with the latest backed up state, ensuring minimal disruption. The combination of network redundancy and database persistence ensures that no single failure mode leaves the system inoperable.

5.8 Drawbacks

Despite its robust design, the elevator system has a few limitations:

- **Emergency Stop State Limitation:** If the elevator stops between floors, the hardware cannot recognize its previous floor due to sensor constraints. A power reset is the only way to clear this state. Otherwise one must go to a different floor before going back to the original floor in order for floor sensor to update without restarting.
- **Managers Disconnection Handling:** If an elevator remains disconnected for too long, the Manager Node assumes it is permanently offline and reallocates its requests. However, it cannot distinguish between a long disconnection and a temporary power failure. This means that we process the same queue of requests twice independently.

Despite these drawbacks, the system is highly resilient and ensures that no requests are lost. Through a combination of networked management, failover processes, and database recovery, the elevator will always complete its assigned queue, even in the event of failures.

6 Results

The system we designed and implemented performed exceptionally well, demonstrating fault tolerance, scalability, and efficiency. Our architectural choices played a crucial role in streamlining development, debugging, and optimization. By separating responsibilities into distinct nodes within a distributed system, we created a flexible and maintainable structure that allows for future integration of additional features without significant refactoring. The ability to scale dynamically by adjusting the number of database, manager, and elevator nodes further solidified the systems robustness.

A key factor in achieving this scalability was our use of Zenohs network configuration, which enabled seamless synchronization across nodes while reducing unnecessary network overhead. By leveraging Zenohs capabilities, we ensured real time data consistency, allowing processes to stay synchronized without excessive communication, even in larger deployments. This made managing elevator assignments, state synchronization, and fault tolerance far more efficient than traditional polling based approaches.

Our approach to process execution allowed for easy deployment and scalability. Running additional database or manager nodes was as simple as executing:

```
$ sudo -E DATABASE_NETWORK_ID=<ID> ELEVATOR_NETWORK_ID_LIST="[<ID 1>,<ID
    2>,...,<ID N>]" NUMBER_FLOORS=<NUMBER_FLOORS> cargo run --bin
    database_process_pair
```

Similarly, manager nodes could be scaled with:

```
$ sudo -E MANAGER_ID=<ID> ELEVATOR_NETWORK_ID_LIST="[<ID 1>,<ID 2>,...,<ID N>]"
    cargo run --bin manager_process_pair
```

For elevator nodes, each instance required specifying a unique hardware port and floor count:

```
$ sudo -E ELEVATOR_NETWORK_ID=<ID> ELEVATOR_HARDWARE_PORT=<PORT> NUMBER_FLOORS=<
    NUMBER_FLOORS> cargo run --bin elevator_process_pair
```

These flexible configurations allowed us to rapidly test different setups, ensuring the system could handle various scenarios efficiently. The use of environment variables and script automation in **".bash"** files further simplified deployment, making it straightforward to modify network settings, hardware ports, and elevator configurations without requiring changes to the source code.

Once the system was set up, we could modify **".bash"** files and use them as macros to simplify execution. Running the system was as straightforward as:

```
$ cd elevator-server
$ ./run_server.bash
```

Then, in a new terminal, start the database process:

```
$ ./run_database.bash <ID>
```

Next, in another terminal, launch the manager process:

```
$ ./run_manager.bash <ID>
```

Finally, start the elevator process in a separate terminal:

```
$ ./run_elevator.bash <ID>
```

This method streamlined the execution of the system, allowing easy customization and ensuring that each process ran in an isolated environment for better fault tolerance and debugging.

The fault tolerance mechanisms proved to be highly effective. When an elevator node failed, the manager detected the missing heartbeat signal and redistributed requests among available elevators, preventing service disruptions. Similarly, database nodes functioned as a distributed network, meaning any node could temporarily assume leadership in the event of a failure. This ensured that data integrity and synchronization were maintained without requiring manual intervention.

By implementing cooperative scheduling, state machines, and real time event handling, we created a system that efficiently handled dynamic request assignments while minimizing idle time. The real time responsiveness of the system was evident, as elevators adjusted their movements dynamically based on incoming requests and state transitions.

At the time of completing this report in mid-February, we had successfully implemented and tested the full system within a 1 month time frame, significantly ahead of the approximate 3 month project deadline. This rapid development was made possible by a well structured architecture that allowed for parallel work on different components, clear separation of concerns, and a streamlined debugging process. The modular design ensures that the system can be extended in the future without requiring extensive rework, making it a scalable and adaptable solution for real time elevator control in distributed environments.

7 Conclusion

This project demonstrated the successful implementation of a fault tolerant, scalable, and efficient real time elevator system. Through a well structured distributed architecture, we efficiently managed synchronization, scheduling, and fault recovery, ensuring smooth operation under various conditions. The use of multi threaded processes, cooperative scheduling, and real time state machines enabled a responsive and reliable system.

One of the key takeaways was the importance of designing a scalable and modular architecture from the start. The decision to separate database, manager, and elevator nodes significantly improved maintainability, debugging, and future expansion. Using Zenoh as a distributed network layer and automating deployment through .bash scripts streamlined system execution and testing.

The flexibility and control that Rust provided was a significant advantage, particularly in memory safety and concurrency management. Even as beginners, we found Rust to be a great language for building a robust system. Despite its syntax heavy nature, we successfully navigated its complexities with some help from ChatGPT, allowing us to implement a well structured and efficient solution.

Overall, this project was highly rewarding. We gained hands on experience in real time systems, distributed computing, and multi threaded programming, reinforcing the importance of structured design, automation, and fault tolerant architecture in complex systems. While some aspects, such as carefully planning the best approach for system architecture, required considerable time and effort, the final system performed reliably, meeting all mandatory and bonus requirements efficiently.

Our GitHub repository with the whole project can be found here[5]

References

[1] Zetta Scale. *Zenoh: Data-Centric Communication for the Cloud-to-Things Continuum*. Accessed: January 2025. 2022. URL: https://conferences2.sigcomm.org/acm-icn/2022/assets/zenoh-1-Opening-5f566872e781cac4aa9e0460e26151e742b675ead58b9bc2db2bf72b5d78ca88.pdf.

[2] Angelo Corsaro et al. *Zenoh: Unifying Communication, Storage and Computation from the Cloud to the Microcontroller*. Accessed: January 2025.

[3] Wen-Yew Liang, Yuyuan Yuan, and Hsiang-Jui Lin. *A Performance Study on the Throughput and Latency of Zenoh, MQTT, Kafka, and DDS*. Accessed: January 2025. 2023. URL: https://arxiv.org/pdf/2303.09419.

[4] Object Management Group. *What is DDS*. Accessed: January 2025. URL: https://www.dds-foundation.org/what-is-dds-3/.

[5] Martynas Smilingis, Anders Jervan, and Andrine Lunde Thoresen. *TTK4145 Real-Time Programming*. Accessed: January 2025. 2025. URL: https://github.com/PizzaAllTheWay/TTK4145_Real_Time_Programming/.