Innhald frå Rust-filer

Fil: 3701d923/TTK4145_Real_Time_Programming-main/src/main.rs

```
fn main() {}
```

Innhald frå Rust-filer

Fil: 3701d923/TTK4145_Real_Time_Programming-main/src/lib.rs

```
pub mod elevator_io {
    pub mod data;
    pub mod driver;
    pub mod driver_sync;
}

pub mod elevator_logic {
    pub mod state_machine;
    pub mod utils;
}

pub mod distributed_systems {
    pub mod utils;
}

pub mod elevator_algorithm {
    pub mod cost_algorithm;
    pub mod utils;
}
```

Innhald frå Rust-filer

Fil: 3701d923/TTK4145_Real_Time_Programming-main/src/database.rs

```
// * NOTE: Some networking libraries requiring root privileges (for eks for pinging to router to read if we are still on the same network)
// * NOTE: Every new database needs its own unique DATABASE_ID
// *
// * Because of these factors we must run this process as follows:
// * $ sudo -E DATABASE_NETWORK_ID=<ID> ELEVATOR_NETWORK_ID_LIST="[<ID 1>,<ID 2>,...,<ID N>]" NUMBER_FLOORS=<NUMBER FLOORS> cargo run --bin database

// Library that allows us to use environment variables or command-line arguments to pass variables from terminal to the program directly
use std::env;

// Libraries for multithreading in cooperative mode
use std::sync::Arc;
use tokio::sync::{watch, Mutex, RwLock}; // For optimization RwLock for data that is read more than written to. // Mutex for 1-1 ratio of read write to
use tokio::time::{sleep, Duration};

// Libraries for highly customizable distributed network mode setup
use std::fs;
use std::path::Path;
use zenoh::Config;

// Libraries for distributed network
use zenoh::open;

// Libraries for network status and diagnostics
// * NOTE: Because of some functions in utils library that uses ICMP sockets, witch are ONLY available for root user, we must run our program as sudo cargo run
use elevator_system::distributed_systems::utils::{get_router_ip, network_status, parse_message_to_hashset_u8, parse_message_to_string, wait_with_timeout, NetworkStatus};

// Libraries for constructing data structures that are thread safe
use elevator_system::elevator_algorithm::utils::{AlgoInput, ElevState};
use elevator_system::elevator_logic::utils::ElevHallRequests;
use once_cell::sync::Lazy;
use std::collections::{HashMap, HashSet};

// Global Variable ----------
const SYNC_INTERVAL: u64 = 1000; // ms
const NETWORK_CHECK_INTERVAL: u64 = 5000; // ms

const LEADER_TOPIC: &str = "sync/database/leader";

// Set up environment variables ----------
// Get the DATABASE_NETWORK_ID from the environment variable, defaulting to 0 if not set
// !NOTE: Every new Rust process needs their own unique DATABASE_NETWORK_ID
static DATABASE_NETWORK_ID: Lazy<i64> = Lazy::new(|| {
    env::var("DATABASE_NETWORK_ID")
        .unwrap_or_else(|_| "0".to_string())
        .parse()
        .expect("DATABASE_NETWORK_ID must be a valid integer")
});

// Get the ELEVATOR_NETWORK_ID_LIST from the environment variable
// Defaulting to [0] if not set
static ELEVATOR_NETWORK_ID_LIST: Lazy<Vec<i64>> = Lazy::new(|| {
    // Expect the variable in the form "[1,2,3]"
    let list_str = env::var("ELEVATOR_NETWORK_ID_LIST").unwrap_or_else(|_| "[0]".to_string());
    list_str
        .trim_matches(|c| c == '[' || c == ']')
        .split(',')
        .map(|s| s.trim().parse().expect("Invalid elevator id in list"))
        .collect()
});

// Create a static parameter for number of floors this specific elevator serves
// If none => Default to NUMBER FLOORS: 4
static NUMBER_FLOORS: Lazy<u8> = Lazy::new(|| {
    env::var("NUMBER_FLOORS")
        .unwrap_or_else(|_| "4".to_string())
        .parse()
        .expect("NUMBER_FLOORS must be a valid integer")
});

// Data Structure Construction ----------
// NOTE: Box::leak() is a powerful yet dangerous command
// NOTE: Used inappropriately in a dynamic continuous running process, it will clog up the memory as it is never deallocated, causing memory leaks and overflows
// NOTE: However for use in startup where the values will never be manipulated afterwards this is a safe way to us it in
// NOTE: Since we only manipulate memory on startup for Topics, we don't have to worry about memory leaks and overflows :)
type SharedData = Arc<Mutex<String>>; // All shared data is stored as a String at the end of the day

#[derive(Clone)]
struct DataStreamConfig {
    temp_topic: &'static str,
    stor_topic: &'static str,
    shared_data: SharedData,
    rebroadcast_interval: u64, // * [ms] TIPS: Setting this value to 0 disables rebroadcasting ability for that specific datastream
}

static DATA_STREAMS_ELEVATOR: Lazy<Vec<DataStreamConfig>> = Lazy::new(|| {
    ELEVATOR_NETWORK_ID_LIST
        .iter()
        .flat_map(|&id| {
            let id_str = id.to_string();
            vec![
                // This topic MUST be broadcasted
                // Reason being is that database itself uses it as backup
                // If a database dies and then rejoins,
                // by rebroadcasting this data, we get to read the states of the elevators on initialization again
                // This way we hold everything synchronized and backed up
                DataStreamConfig {
                    temp_topic: Box::leak(format!("temp/elevator{}/states", id_str).into_boxed_str()),
                    stor_topic: Box::leak(format!("stor/elevator{}/states", id_str).into_boxed_str()),
                    shared_data: Arc::new(Mutex::new(String::new())),
                    rebroadcast_interval: SYNC_INTERVAL, // ms
                },
                // Heartbeat topic
                DataStreamConfig {
                    temp_topic: Box::leak(format!("temp/elevator{}/heartbeat", id_str).into_boxed_str()),
                    stor_topic: Box::leak(format!("stor/elevator{}/heartbeat", id_str).into_boxed_str()),
                    shared_data: Arc::new(Mutex::new(String::new())),
                    rebroadcast_interval: 0, // DISABLED
                },
                // These topics must be rebroadcasted
                // Reason being is that node initialization depend on data that is backed up
                // The way nodes receive backup data is listen to stor/ topics for next rebroadcast
                DataStreamConfig {
                    temp_topic: Box::leak(format!("temp/elevator{}/backup", id_str).into_boxed_str()),
                    stor_topic: Box::leak(format!("stor/elevator{}/backup", id_str).into_boxed_str()),
                    shared_data: Arc::new(Mutex::new(String::new())),
                    rebroadcast_interval: 500, // ms (NOTE: Should be the same as BACKUP_INTERVAL in "elevator.rs")
                },
            ]
        })
        .collect()
});

static DATA_STREAMS_MANAGER: Lazy<Vec<DataStreamConfig>> = Lazy::new(|| {
    vec![DataStreamConfig {
        temp_topic: Box::leak(format!("temp/manager/request").into_boxed_str()),
        stor_topic: Box::leak(format!("stor/manager/request").into_boxed_str()),
        shared_data: Arc::new(Mutex::new(String::new())),
        rebroadcast_interval: 0, // DISABLED
    }]
});

const HALL_REQUESTS_SYNC: &str = "sync/elevator/hall/requests";
const HALL_REQUESTS_UP_STOR: &str = "stor/elevator/hall/requests/up";
const HALL_REQUESTS_DOWN_STOR: &str = "stor/elevator/hall/requests/down";
const HALL_REQUESTS_BACKUP_INTERVAL: u64 = 500; // ms

const ELEVATOR_DATA_SYNC: &str = "sync/elevator/data/synchronized";

#[tokio::main]
async fn main() {
                                         //        Distributed        Network        Initialization        (START)
============================================================================================
=========
    println!("DATABASE_NETWORK_ID: {}", *DATABASE_NETWORK_ID);
    println!("ELEVATOR_NETWORK_ID_LIST: {:#?}", *ELEVATOR_NETWORK_ID_LIST);
```

Innhald frå Rust-filer

```
    println!("NUMBER_FLOORS: {}", *NUMBER_FLOORS);
    println!();

    // Specify path to highly customable network modes for distributed networks
    // Most important settings: peer-2-peer and scouting to alow multicast and robust network connectivity
    // Then Load configuration from JSON5 file
    // Finally initialize networking session
    let networking_config_path = Path::new("network_config.json5");

            let networking_config_data = fs::read_to_string(networking_config_path).expect("Failed to read the
network_config.json5 file");
     let config: Config = Config::from_json5(&networking_config_data).expect("Failed to parse the network_config.json5
file");

    let network_session = open(config).await.expect("Failed to open Zenoh session");
                                          // Distributed Network Initialization (STOP)
==========================================================================================
=========

                                                  // Database Initialization (START)
==========================================================================================
=========
    // Initialization step to check if stored messages were updated while this database node was gone
    // If we detect new stored data we update our internal data to match outside world
    let mut tasks = Vec::new();

    // Add together all streaming topics into 1 big vector array
    let mut all_data_streams: Vec<DataStreamConfig> = Vec::new();
    all_data_streams.extend(DATA_STREAMS_ELEVATOR.iter().cloned()); // Clone the data from Lazy
    all_data_streams.extend(DATA_STREAMS_MANAGER.iter().cloned()); // Clone the data from Lazy

    for stream in all_data_streams {
        let stor_topic = stream.stor_topic;
        let shared_data = stream.shared_data.clone();
        let networking_session_clone = network_session.clone();

        tasks.push(tokio::spawn(async move {
            let subscriber_stor = networking_session_clone
                .declare_subscriber(stor_topic)
                .await
                .expect("Failed to declare stor subscription topic");

            // Wait for some stored data
            // If we don't get any in a certain amount of time we assume there is no stored data, so we pass
            // If we find stored data being published we save it internally in our data base
            // 10 000 ms because it takes some time for network config to configure our networking protocol, thus we need to
compensate for it
            // + wait a bit for a given broadcast interval just to be sure
            let init_timeout = 10000 + stream.rebroadcast_interval;
            let result = wait_with_timeout(init_timeout, subscriber_stor.recv_async()).await;

            if let Some(message) = result {
                let data_new = parse_message_to_string(message);
                println!("New data for storage: {}: {}", stor_topic, data_new);

                let mut data = shared_data.lock().await;
                *data = data_new.to_string();
            } else {
                println!("No new data for storage: {}", stor_topic);
            }
        }));
    }

    for task in tasks {
        let _ = task.await;
    }
                                        // Database Initialization (STOP)
============================================================================================
=========

                            // Elevator Data Synchronization Initialization (START)
============================================================================================
=========
    // Elevator State Backup ----------
    // Stores elevator states in a **shared HashMap** (`elevator_states`)
    // Uses **RwLock** since reads will be more frequent than writes

    // Shared resource for storing elevator states
    // - `RwLock` allows multiple readers and a single writer (better performance for read-heavy workloads)
    let elevator_states: Arc<RwLock<HashMap<i64, String>>> = Arc::new(RwLock::new(HashMap::new()));

    // Before subscribing to updates, we initialize the HashMap with existing state data
    // This ensures that the database starts with **correct values**
    // Extracts initial state from the `DATA_STREAMS_ELEVATOR` list and maps it to each elevator ID
    let elevator_states_clone = elevator_states.clone();
    {
        let mut elevator_states = elevator_states_clone.write().await;
        let mut index = 0; // Tracks valid elevator IDs

        for stream in DATA_STREAMS_ELEVATOR.iter() {
            if stream.temp_topic.contains("/states") {
                if let Some(&id) = ELEVATOR_NETWORK_ID_LIST.get(index) {
                    let initial_state = stream.shared_data.lock().await.clone(); // Extract initial state
                    elevator_states.insert(id, initial_state);
                    index += 1; // Only increment if we successfully mapped an elevator
                }
            }
        }
    }

    // Hall Requests Backup ----------
    // The only thing we need to update now are hall requests
    // We subscribe to hall requests backup topics and listen to them for a moment
    // If no new hall calls we just continue with no new data
    // If there are responses, we back that data up

    // Shared resources: Separate HashSets for UP and DOWN hall requests
    let hall_requests_up: Arc<RwLock<HashSet<u8>>> = Arc::new(RwLock::new(HashSet::new()));
    let hall_requests_down: Arc<RwLock<HashSet<u8>>> = Arc::new(RwLock::new(HashSet::new()));

    // Create backup storage subscribers
    let backup_hall_requests_up_subscriber = network_session
        .declare_subscriber(HALL_REQUESTS_UP_STOR)
        .await
        .expect("Failed to declare UP requests publisher");
    let backup_hall_requests_down_subscriber = network_session
        .declare_subscriber(HALL_REQUESTS_DOWN_STOR)
        .await
        .expect("Failed to declare DOWN requests publisher");

    // Create tasks to get backup data if it exists
    // Wait for some stored data
    // If we don't get any in a certain amount of time we assume there is no stored data, so we pass
    // If we find stored data being published we save it internally in our data base
    // 5 000 ms because it takes some time for network config to configure our networking protocol, thus we need to compensate for it
    // + wait a bit for a given broadcast interval just to be sure
    let mut tasks = Vec::new();
    let backup_init_timeout = 5000 + HALL_REQUESTS_BACKUP_INTERVAL;

    let hall_requests_up_clone = hall_requests_up.clone();
    tasks.push(tokio::spawn(async move {
        let result = wait_with_timeout(backup_init_timeout, backup_hall_requests_up_subscriber.recv_async()).await;

        if let Some(message) = result {
            let data_new: HashSet<u8> = parse_message_to_hashset_u8(message);
            println!("New data from: {}: {:#?}", HALL_REQUESTS_UP_STOR, data_new);

            let mut data = hall_requests_up_clone.write().await;
            *data = data_new;
        } else {
            println!("No new data from: {}", HALL_REQUESTS_UP_STOR);
        }
    }));

    let hall_requests_down_clone = hall_requests_down.clone();
    tasks.push(tokio::spawn(async move {
        let result = wait_with_timeout(backup_init_timeout, backup_hall_requests_down_subscriber.recv_async()).await;

        if let Some(message) = result {
            let data_new: HashSet<u8> = parse_message_to_hashset_u8(message);
            println!("New data from: {}: {:#?}", HALL_REQUESTS_DOWN_STOR, data_new);

            let mut data = hall_requests_down_clone.write().await;
            *data = data_new;
        } else {
            println!("No new data from: {}", HALL_REQUESTS_DOWN_STOR);
        }
    }));

    for task in tasks {
        let _ = task.await;
    }
                              // Elevator Data Synchronization Initialization (STOP)
============================================================================================
=========

                                                   // Network Monitoring (START)
============================================================================================
=========
    // Spawn a separate task to check network status every so often
    // If we detect we have been disconnected from the network we kill ourselves
    tokio::spawn(async move {
        let router_ip = match get_router_ip().await {
            Some(ip) => ip,
            None => {
                println!("#======================================#");
                println!("ERROR: Failed to retrieve the router IP");
                println!("Killing myself...");
                println!("Gugu gaga *O*");
                println!("#======================================#");
                std::process::exit(1);
            }
        };

        loop {
            match network_status(router_ip).await {
                NetworkStatus::Connected => {
                    // Do nothing
                }
                NetworkStatus::Disconnected => {
                    println!("#======================================#");
                    println!("ERROR: Disconnected from the network!");
                    println!("Killing myself...");
                    println!("Shiding and crying T_T");
                    println!("#======================================#");
                    std::process::exit(1);
                }
            }

            sleep(Duration::from_millis(NETWORK_CHECK_INTERVAL)).await;
        }
    });
                                                   // Network Monitoring (STOP)
============================================================================================
=========

                                                           // Synchronization (START)
```

Innhald frå Rust-filer

```
==========================================================================================
    let leader_publisher = network_session.declare_publisher(LEADER_TOPIC).await.expect("Failed to declare leader publisher");

    let leader_subscriber = network_session.declare_subscriber(LEADER_TOPIC).await.expect("Failed to declare leader subscriber");

    let leader = Arc::new(RwLock::new(false));

    // Leader monitoring task ----------
    {
        let leader = leader.clone();
        let leader_elect_interval = SYNC_INTERVAL * 5; // 5x sync because we want to make sure everyone who wants to be a leader has broadcasted it at least once

        tokio::spawn(async move {
            loop {
                // Wait for a leader broadcast within the election interval
                let result = wait_with_timeout(leader_elect_interval, leader_subscriber.recv_async()).await;

                // Check the results
                // If we got a time-out, that means no one else on the network wants to be a leader
                // => become default leader
                // If there is someone else trying to become the leader
                // => Chose leader with lowest ID
                if let Some(message) = result {
                    // Parse leader ID from the announcement
                    let id = parse_message_to_string(message);

                    if let Ok(leader_id) = id.parse::<i64>() {
                        let mut are_we_leader = leader.write().await;

                        if leader_id < *DATABASE_NETWORK_ID {
                            *are_we_leader = false; // Step down from leadership
                        } else {
                            *are_we_leader = true; // Become leader
                        }
                    }
                } else {
                    // No leader broadcast received within the timeout
                    let mut is_leader_lock = leader.write().await;
                    *is_leader_lock = true; // Default to becoming the leader

                    println!("No leader detected, becoming the leader (o0o)");
                }
            }
        });
    }

    // Leader broadcasting task ----------
    {
        let leader = leader.clone();

        tokio::spawn(async move {
            loop {
                if *leader.read().await {
                    leader_publisher
                        .put(DATABASE_NETWORK_ID.to_string().as_bytes())
                        .await
                        .expect("Failed to announce leadership");
                }

                sleep(Duration::from_millis(SYNC_INTERVAL)).await;
            }
        });
    }
                                                  //          Synchronization          (STOP)
============================================================================================
=========

                                                    //          Database          (START)
============================================================================================
=========
    // Add together all streaming topics into 1 big vector array
    let mut all_data_streams: Vec<DataStreamConfig> = Vec::new();
    all_data_streams.extend(DATA_STREAMS_ELEVATOR.iter().cloned()); // Clone the data from Lazy
    all_data_streams.extend(DATA_STREAMS_MANAGER.iter().cloned()); // Clone the data from Lazy

    // Data Monitor, Store and Broadcast
    for stream in all_data_streams {
        let temp_topic = stream.temp_topic;
        let stor_topic = stream.stor_topic;
        let shared_data = stream.shared_data.clone();
        let leader = leader.clone();

        let subscriber_temp = network_session
            .declare_subscriber(temp_topic)
            .await
            .expect("Failed to declare temp subscription topic");

        let publisher_stor = network_session.declare_publisher(stor_topic).await.expect("Failed to declare stor publisher");

        tokio::spawn(async move {
            loop {
                // Wait for a new message or timeout depending on the data stream settings
                if stream.rebroadcast_interval != 0 {
                    let result = wait_with_timeout(stream.rebroadcast_interval, subscriber_temp.recv_async()).await;

                    // Process received data
                    if let Some(message) = result {
                        let data_new = parse_message_to_string(message);

                        // Update shared data
                        let mut data = shared_data.lock().await;
                        *data = data_new.to_string();
                    } else {
                        // No new data received or timeout occurred
                        // Do nothing
                    }
                } else {
                    match subscriber_temp.recv_async().await {
                        Ok(message) => {
                            let data_new = parse_message_to_string(message);

                            // Update shared data
                            let mut data = shared_data.lock().await;
                            *data = data_new.to_string();
                        }
                        Err(e) => {
                            // Log an error if receiving a message fails
                            println!("#========================================#");
                            println!("ERROR: Failed to receive data from {}", temp_topic);
                            println!("Error code: {}", e);
                            println!("Killing myself...");
                            println!("ReeeEEEEeeee!");
                            println!("#========================================#");
                            std::process::exit(1);
                        }
                    }
                }

                // Publish the value stored in shared data if this node is the leader
                if *leader.read().await {
                    let data = shared_data.lock().await.clone();
                    if publisher_stor.put(data.as_bytes()).await.is_ok() {
                        // Successfully sent data
                        // Do Nothing

                        // println!("DEBUG: {}: {}", stor_topic, data);
                    } else {
                        // Log an error if sending a message fails
                        println!("#========================================#");
                        println!("ERROR: Failed to send data to {}", stor_topic);
                        println!("Killing myself...");
                        println!("Bruuhhhhhh =-=");
                        println!("#========================================#");
                        std::process::exit(1);
                    }
                }
            }
        });
    }
```

// Database (STOP) ===

Innhald frå Rust-filer

```
                                        // Elevator Data Synchronization (START)
=============================================================================================
=========
    // Set up notifications ----------
    // Notification channel (tx = sender, rx = receiver)
    // When any of the states update, we notify the main synchronization thread through this cannel
    // This way, the final synchronization thread for all the elevators data can do its job
    // It will combine and decide if we should send the data or not
    let (notify_tx, mut notify_rx) = watch::channel(false); // Initial value is `false`, meaning no updates yet

    // Elevator State Sync Threads ----------
    // Instead of polling, this section listens for real-time updates
    // Each elevator gets **its own async task** that waits for new messages
    // When a new state update arrives, it is **immediately** written to the HashMap
    let network_session_clone = network_session.clone();
    let mut index = 0; // Reset index for correct elevator ID mapping

    for stream in DATA_STREAMS_ELEVATOR.iter() {
        if stream.stor_topic.contains("/states") {
            if let Some(&id) = ELEVATOR_NETWORK_ID_LIST.get(index) {
                // Each thread listens to its designated `stor/elevator{id}/states` topic.
                let subscriber = network_session_clone
                    .declare_subscriber(stream.stor_topic)
                    .await
                    .expect("Failed to declare stor subscription topic");

                let elevator_states_clone = elevator_states.clone();
                let id_clone = id; // Copy ID (i64 is Copy, no need for .clone())
                let notify_tx_clone = notify_tx.clone();

                // Runs **forever**, listening for new messages.
                // Updates only the **correct elevator's** state when data arrives.
                tokio::spawn(async move {
                    while let Ok(message) = subscriber.recv_async().await {
                        let new_state = parse_message_to_string(message);

                        let mut elevator_states = elevator_states_clone.write().await;
                        elevator_states.insert(id_clone, new_state.clone());

                        // Notify listeners that an update happened
                        let _ = notify_tx_clone.send(true);

                        // println!("DEBUG: Updated Elevator {} State: {}", id_clone, new_state);
                    }
                });
            }

            index += 1; // Only increment if we successfully mapped an elevator
        }
    }
```

Innhald frå Rust-filer

```rust
// Elevator Hall Request Sync Threads ----------
// Synchronize hall request data
// Very similar to Elevator State Sync Threads
// However here there is only one thread
// Since any elevator can write to this topic it is a synchronization topic of its own
// And the data goes to a HashSet for good data structure

// Subscribe to HALL_REQUESTS_SYNC updates
let hall_requests_up_clone = hall_requests_up.clone();
let hall_requests_down_clone = hall_requests_down.clone();
let notify_tx_clone = notify_tx.clone();

let hall_requests_subscriber = network_session
    .declare_subscriber(HALL_REQUESTS_SYNC)
    .await
    .expect("Failed to subscribe to HALL_REQUESTS_SYNC");

tokio::spawn(async move {
    while let Ok(message) = hall_requests_subscriber.recv_async().await {
        // Convert Zenoh message to a JSON string
        let json_str = parse_message_to_string(message);

        // Attempt to deserialize JSON into `ElevHallRequests`
        if let Ok(request) = serde_json::from_str::<ElevHallRequests>(&json_str) {
            // Add/Remove requests in HashSets based on received data
            if let Some(floor) = request.add_up {
                let mut hall_set_up = hall_requests_up_clone.write().await;
                hall_set_up.insert(floor);
            }
            if let Some(floor) = request.add_down {
                let mut hall_set_down = hall_requests_down_clone.write().await;
                hall_set_down.insert(floor);
            }
            if let Some(floor) = request.remove_up {
                let mut hall_set_up = hall_requests_up_clone.write().await;
                hall_set_up.remove(&floor);
            }
            if let Some(floor) = request.remove_down {
                let mut hall_set_down = hall_requests_down_clone.write().await;
                hall_set_down.remove(&floor);
            }
        } else {
            eprintln!("ERROR: Failed to deserialize Hall Request JSON: {:#?}", json_str);
        }

        // Notify listeners that an update happened
        let _ = notify_tx_clone.send(true);
    }
});

// Elevator Hall Request Backup Threads ----------
// NOTE: Backup only happens if our node is the leader
// We also want to back up Hall Requests that are currently pending
// This is done so in case our database crashes, we can always recover from backup
// Meaning we never lose our previous hall requests
let backup_hall_requests_up_publisher = network_session
    .declare_publisher(HALL_REQUESTS_UP_STOR)
    .await
    .expect("Failed to declare UP requests publisher");
let backup_hall_requests_down_publisher = network_session
    .declare_publisher(HALL_REQUESTS_DOWN_STOR)
    .await
    .expect("Failed to declare DOWN requests publisher");

let hall_requests_up_clone = hall_requests_up.clone();
let hall_requests_down_clone = hall_requests_down.clone();
let leader_clone = leader.clone();

tokio::spawn(async move {
    loop {
        // NOTE: Only backup data if you are the leader
        // If not we just sit and wait
        if *leader_clone.read().await {
            let hall_up = hall_requests_up_clone.read().await;
            let hall_down = hall_requests_down_clone.read().await;

            backup_hall_requests_up_publisher
                .put(format!("{:?}", hall_up).to_string().as_bytes())
                .await
                .expect("Failed to backup Hall Requests UP");
            backup_hall_requests_down_publisher
                .put(format!("{:?}", hall_down).to_string().as_bytes())
                .await
                .expect("Failed to backup Hall Requests DOWN");

            sleep(Duration::from_millis(HALL_REQUESTS_BACKUP_INTERVAL)).await;
        }
    }
});

// Elevator Data Synchronization Thread ----------
// NOTE: Synchronization only happens if our node is the leader
let elevator_data_sync_publisher = network_session
    .declare_publisher(ELEVATOR_DATA_SYNC)
    .await
    .expect("Failed to declare Elevator Data Synchronization publisher");

let elevator_states_clone = elevator_states.clone();
let hall_requests_up_clone = hall_requests_up.clone();
let hall_requests_down_clone = hall_requests_down.clone();
let leader_clone = leader.clone();

tokio::spawn(async move {
    while notify_rx.changed().await.is_ok() {
```

Innhald frå Rust-filer

```
        // NOTE: Only send synchronized data if we are the leader
        // Otherwise we just wait for new change
        if *leader_clone.read().await {
                // Since we are a leader and there was a change we get to combine the data into a single JSON string to output

            let elev_states = elevator_states_clone.read().await;
            let hall_up = hall_requests_up_clone.read().await;
            let hall_down = hall_requests_down_clone.read().await;

            // Format JSON -----
            // Read elevator states
            let mut formatted_elevators: HashMap<String, ElevState> = HashMap::new();

            for (&id, state_json) in elev_states.iter() {
                if let Ok(state) = serde_json::from_str::<ElevState>(state_json) {
                    formatted_elevators.insert(id.to_string(), state);
                }
            }

            // Read hall requests
            let mut hall_requests_2d = vec![vec![false, false]; (*NUMBER_FLOORS).into()];

            // UP
            for &floor in hall_up.iter() {
                if floor < *NUMBER_FLOORS {
                    hall_requests_2d[floor as usize][1] = true;
                }
            }

            // DOWN
            for &floor in hall_down.iter() {
                if floor < *NUMBER_FLOORS {
                    hall_requests_2d[floor as usize][0] = true;
                }
            }

            // Construct the final system state
            let system_state = AlgoInput { hallRequests: hall_requests_2d, states: formatted_elevators };

            // Convert it into JSON format
            let json_output = serde_json::to_string_pretty(&system_state).expect("Failed to serialize system state");

            // println!("DEBUG: Json: {:#?}", json_output);

            // Send JSON to manager node
            elevator_data_sync_publisher
                .put(json_output.as_bytes())
                .await
                .expect("Failed to publish Elevator Data Synchronization");
        }
    }
    });
    // Elevator Data Synchronization (STOP)
============================================================================================
=========
    // Keep the program running
    loop {
        tokio::task::yield_now().await; // Yield to other tasks
    }
}
```

Innhald frå Rust-filer

Fil: 3701d923/TTK4145_Real_Time_Programming-main/src/elevator.rs

```
// * NOTE: We don't actually need to run the node in "sudo", however for consistency with other nodes that require "sudo" we have made this node also run in "sudo"
// * NOTE: Every new elevator needs its own unique ELEVATOR_NETWORK_ID and ELEVATOR_HARDWARE_PORT
// *
// * Because of these factors we must run this process as follows:
// * $ sudo -E ELEVATOR_NETWORK_ID=<ID> ELEVATOR_HARDWARE_PORT=<PORT> NUMBER_FLOORS=<NUMBER FLOORS> cargo run --bin elevator

// Library that allows us to use environment variables or command-line arguments to pass variables from terminal to the program directly
use std::env;

// Libraries for real time systems
use std::collections::HashSet;
use std::sync::Arc;

use tokio::spawn;
use tokio::sync::{mpsc, RwLock};
use tokio::task::yield_now;
use tokio::time::{self, Duration};

// Libraries for highly customizable distributed network mode setup
use std::fs;
use std::path::Path;
use zenoh::Config;

// Libraries for distributed network
use zenoh::open;

// Libraries for network status and diagnostics
// * NOTE: Because of some functions in utils library that uses ICMP sockets, witch are ONLY available for root user, we must run our program as sudo cargo run
use elevator_system::distributed_systems::utils::{get_router_ip, network_status, parse_message_to_elevator_backup, parse_message_to_elevator_requests, wait_with_timeout, ElevatorBackup, NetworkStatus};

// Import necessary drivers for controlling the elevator
use elevator_system::elevator_io::{data, driver};
use elevator_system::elevator_logic::state_machine;
use elevator_system::elevator_logic::utils::{create_hall_request_json, Direction, State};

// Import elevator manager algorithm library because when sending data we have to format our data in a way that other manager algorithm nodes can understand it
use elevator_system::elevator_algorithm::utils::ElevState;

// Library for constructing data structures that are thread safe
use once_cell::sync::Lazy;

// Global Variable ----------
const NETWORK_CHECK_INTERVAL: u64 = 5000; // ms
const POLL_INTERVAL: u64 = 200; // ms
```

Innhald frå Rust-filer

```
const HEARTBEAT_INTERVAL: u64 = 1000; // ms
const BACKUP_INTERVAL: u64 = 500; // ms

// Topics for datastream ----------
const HALL_REQUESTS_SYNC_TOPIC: &str = "sync/elevator/hall/requests";

const MANAGER_TOPIC: &str = "stor/manager/request";

struct Topics {
    heartbeat: &'static str,

    elevator_states: &'static str,

    backup_temp: &'static str,
    backup_stor: &'static str,
}

// NOTE: Box::leak() is a powerful yet dangerous command
// NOTE: Used inappropriately in a dynamic continuous running process, it will clog up the memory as it is never deallocated, causing memory leaks and overflows
// NOTE: However for use in startup where the values will never be manipulated afterwards this is a safe way to us it in
// NOTE: Since we only manipulate memory on startup for Topics, we don't have to worry about memory leaks and overflows :)
impl Topics {
    fn new(elevator_id: i64) -> Self {
        let id = elevator_id.to_string();
        Topics {
            heartbeat: Box::leak(format!("temp/elevator{}/heartbeat", id).into_boxed_str()),

            elevator_states: Box::leak(format!("temp/elevator{}/states", id).into_boxed_str()),

            backup_temp: Box::leak(format!("temp/elevator{}/backup", id).into_boxed_str()),
            backup_stor: Box::leak(format!("stor/elevator{}/backup", id).into_boxed_str()),
        }
    }
}

// Set up environment variables ----------
// Create a static parameter for number of floors this specific elevator serves
// If none => Default to NUMBER FLOORS: 4
static NUMBER_FLOORS: Lazy<u8> = Lazy::new(|| {
    env::var("NUMBER_FLOORS")
        .unwrap_or_else(|_| "4".to_string())
        .parse()
        .expect("NUMBER_FLOORS must be a valid integer")
});

// Create a static parameter for the hardware address using the port from the environment
// If none => Default to PORT: localhost:15657
// !NOTE: Every new Rust process needs their own unique ELEVATOR_HARDWARE_PORT
static ELEVATOR_HARDWARE_PORT: Lazy<&'static str> = Lazy::new(|| {
    // Read the port from env, defaulting to "15657"
    let port = env::var("ELEVATOR_HARDWARE_PORT").unwrap_or_else(|_| "15657".to_string());
    // Build the address and leak it to get a &'static str.
    Box::leak(format!("localhost:{}", port).into_boxed_str())
});

// Existing topics static block remains, now printing the hardware address as well
// If none => Default to ID: 0
// !NOTE: Every new Rust process needs their own unique ELEVATOR_NETWORK_ID
static ELEVATOR_NETWORK_ID: Lazy<i64> = Lazy::new(|| {
    env::var("ELEVATOR_NETWORK_ID")
        .unwrap_or_else(|_| "0".to_string())
        .parse()
        .expect("ELEVATOR_NETWORK_ID must be a valid integer")
});

// Build topics with our ELEVATOR_NETWORK_ID
static TOPICS: Lazy<Topics> = Lazy::new(|| Topics::new(*ELEVATOR_NETWORK_ID));

#[tokio::main]
async fn main() -> tokio::io::Result<()> {
                                        // Elevator Initialization (START)
============================================================================================
=========
    println!("ELEVATOR_NETWORK_ID: {}", *ELEVATOR_NETWORK_ID);
    println!("ELEVATOR_HARDWARE_PORT: {}", *ELEVATOR_HARDWARE_PORT);
    println!("NUMBER_FLOORS: {}", *NUMBER_FLOORS);
    println!();

    // Start elevator
    let elevator = driver::Elevator::init(*ELEVATOR_HARDWARE_PORT, *NUMBER_FLOORS).await;
    println!("Elevator initialized:\n{:#?}", elevator);
    println!("Jipppyyyyyy!");
    println!();

    // Start by turn off all the lights
    // The lights that should be on will turn on eventually after the backup data kicks in
    let elevator_clone = elevator.clone();
    spawn(async move {
        for floor in 0..*NUMBER_FLOORS {
            elevator_clone.call_button_light(floor, 0, false).await; // Turn OFF UP light
            elevator_clone.call_button_light(floor, 1, false).await; // Turn OFF DOWN light
            elevator_clone.call_button_light(floor, 2, false).await; // Turn OFF CAB light
        }
    });
                                        // Elevator Initialization (STOP)
============================================================================================
=========

                              // Distributed Network Initialization (START)
============================================================================================
=========
    // Specify path to highly customable network modes for distributed networks
```

Innhald frå Rust-filer

```
    // Most important settings: peer-2-peer and scouting to alow multicast and robust network connectivity
    // Then Load configuration from JSON5 file
    // Finally initialize networking session
    let networking_config_path = Path::new("network_config.json5");

            let networking_config_data = fs::read_to_string(networking_config_path).expect("Failed to read the
network_config.json5 file");
     let config: Config = Config::from_json5(&networking_config_data).expect("Failed to parse the network_config.json5
file");

    let network_session = open(config).await.expect("Failed to open Zenoh session");
                                                            // Distributed Network Initialization (STOP)
======================================================================================
=========

                                        // GET - NETWORK: Data backup Initialization (START)
======================================================================================
=========
    // Shared states initialization
    let state = Arc::new(RwLock::new(State::Idle));
    let direction = Arc::new(RwLock::new(Direction::Stop));
    let current_floor = Arc::new(RwLock::new(None::<u8>));
    let cab_queue = Arc::new(RwLock::new(HashSet::new()));
    let hall_up_queue = Arc::new(RwLock::new(HashSet::new()));
    let hall_down_queue = Arc::new(RwLock::new(HashSet::new()));

    // Create backup storage subscribers
    let backup_subscriber = network_session
        .declare_subscriber(TOPICS.backup_stor)
        .await
        .expect("Failed to declare Backup subscriber");

    // Create tasks to get backup data if it exists
    // Wait for some stored data
    // If we don't get any in a certain amount of time we assume there is no stored data, so we pass
    // If we find stored data being published we save it internally in our data base
     // 5 000 ms because it takes some time for network config to configure our networking protocol, thus we need to
compensate for it
    // + wait a bit for a given broadcast interval just to be sure
    let mut tasks = Vec::new();
    let backup_init_timeout = 10000 + BACKUP_INTERVAL;

    let state_clone = state.clone();
    let direction_clone = direction.clone();
    let current_floor_clone = current_floor.clone();
    let cab_queue_clone = cab_queue.clone();
    let hall_up_queue_clone = hall_up_queue.clone();
    let hall_down_queue_clone = hall_down_queue.clone();
    tasks.push(tokio::spawn(async move {
        let result = wait_with_timeout(backup_init_timeout, backup_subscriber.recv_async()).await;

        if let Some(message) = result {
            if let Some(backup_data) = parse_message_to_elevator_backup(message) {
                println!("New data from: {}: {:#?}", TOPICS.backup_stor, backup_data);

                // Save data to the correct location
                {
                    let mut data = state_clone.write().await;
                    *data = backup_data.state;
                }
                {
                    let mut data = direction_clone.write().await;
                    *data = backup_data.direction;
                }
                {
                    let mut data = current_floor_clone.write().await;
                    *data = backup_data.current_floor;
                }
                {
                    let mut data = cab_queue_clone.write().await;
                    *data = backup_data.cab_queue;
                }
                {
                    let mut data = hall_up_queue_clone.write().await;
                    *data = backup_data.hall_up_queue;
                }
                {
                    let mut data = hall_down_queue_clone.write().await;
                    *data = backup_data.hall_down_queue;
                }
            } else {
                println!("Failed to parse data from: {}", TOPICS.backup_stor);
            }
        } else {
            println!("No new data from: {}", TOPICS.backup_stor);
        }
    }));

    for task in tasks {
        let _ = task.await;
    }
                          // GET - NETWORK: Data backup Initialization (START)
===========================================================================
=========

                                                  // Network Monitoring (START)
===========================================================================
=========
    // Spawn a separate task to check network status every so often
    // If we detect we have been disconnected from the network we signal it by changing the shared resource for network
state
    let on_the_network = Arc::new(RwLock::new(true));

    let on_the_network_clone = on_the_network.clone();
```

Innhald frå Rust-filer

```
    tokio::spawn(async move {
        let router_ip = match get_router_ip().await {
            Some(ip) => ip,
            None => {
                println!("#========================================#");
                println!("ERROR: Failed to retrieve the router IP");
                println!("Killing myself...");
                println!("Jinkies (=o.o=)");
                println!("#========================================#");
                std::process::exit(1);
            }
        };

        loop {
            match network_status(router_ip).await {
                NetworkStatus::Connected => {
                    let mut on_the_network = on_the_network_clone.write().await;
                    *on_the_network = true;
                }
                NetworkStatus::Disconnected => {
                    println!("WARNING: Disconnected from the network!");

                    let mut on_the_network = on_the_network_clone.write().await;
                    *on_the_network = false;
                }
            }

            tokio::time::sleep(Duration::from_millis(NETWORK_CHECK_INTERVAL)).await;
        }
    });
                                                            //              Network              Monitoring              (STOP)
===============================================================================================
=========

                                        //           6           -           READING:           Button           orders           (START)
===============================================================================================
=========
    // Create publisher that send hall button requests UP/DOWN to add them to the manager later
    let hall_requests_publisher = network_session
        .declare_publisher(HALL_REQUESTS_SYNC_TOPIC)
        .await
        .expect("Failed to declare Hall Requests publisher");

    // Create a channel for button call updates
    let (button_tx, mut button_rx) = mpsc::channel(32);

    // Poll button calls and send updates
    let elevator_clone = elevator.clone();
    spawn(async move {
        data::call_buttons(elevator_clone, button_tx, Duration::from_millis(POLL_INTERVAL)).await;
    });
    // Process button call updates and update states
    // If its Floor call, we instead send it to the distributed network for manager node decide
    let cab_queue_clone = cab_queue.clone();
    spawn(async move {
        while let Some(button) = button_rx.recv().await {
            match button.call {
                2 => {
                    // CAB button
                    {
                        let mut cab = cab_queue_clone.write().await;
                        cab.insert(button.floor);
                    }
                }
                1 => {
                    // DOWN button
                    {
                        let request = create_hall_request_json(None, Some(button.floor), None, None);

                        hall_requests_publisher.put(request.as_bytes()).await.expect("Failed to publish hall button DOWN");
                    }
                }
                0 => {
                    // UP button
                    {
                        let request = create_hall_request_json(Some(button.floor), None, None, None);

                        hall_requests_publisher.put(request.as_bytes()).await.expect("Failed to publish hall button UP");
                    }
                }
                _ => {
                    // Ignore invalid types
                }
            }
        }
    });
                                        // 6 - READING: Button orders (STOP)
==============================================================================================
=========

                                        // 7 - READING: Floor sensor (START)
==============================================================================================
=========
    // Create a channel for hall sensor updates
    let (floor_tx, mut floor_rx) = mpsc::channel(32);

    // Poll hall sensor and send updates
    let elevator_clone = elevator.clone();
    spawn(async move {
        data::floor_sensor(elevator_clone, floor_tx, Duration::from_millis(50)).await;
    });

    // Process hall updates and print state
    let current_floor_clone = current_floor.clone();
    spawn(async move {
        while let Some(floor) = floor_rx.recv().await {
            // Check if the new value is different or if the current value is None
            let current_floor = *current_floor_clone.read().await;

            if current_floor.is_none() || current_floor != Some(floor) {
                let mut current_floor = current_floor_clone.write().await;
                *current_floor = Some(floor);
            }
        }
    });
                    // 7 - READING: Floor sensor (STOP)
============================================================================================
=========

                    // 8 - READING: Stop button (START)
============================================================================================
=======
    // Shared state for stop button light
    let stop_button_state = Arc::new(RwLock::new(false)); // Initially, stop button is NOT pressed
    let stop_button_state_clone = Arc::clone(&stop_button_state);

    // Control stop button light
    let elevator_clone = elevator.clone();
    let (stop_tx, mut stop_rx) = mpsc::channel(32);

    // Spawn a task to poll the stop button state
    spawn(async move {
        data::stop_button(elevator_clone.clone(), stop_tx, Duration::from_millis(POLL_INTERVAL)).await;
    });

    // Spawn a task to update the stop button state
    spawn(async move {
        while let Some(stop_button) = stop_rx.recv().await {
            {
                let mut stop_state = stop_button_state_clone.write().await;
                *stop_state = stop_button;
            }
        }
    });
                    // 8 - READING: Stop button (STOP)
============================================================================================
=======

                    // 9 - READING: Obstruction switch (START)
============================================================================================
=======
    // Control obstruction switch state
    let obstruction_state = Arc::new(RwLock::new(false)); // Shared state for obstruction switch
    let obstruction_state_clone = Arc::clone(&obstruction_state);
    let elevator_clone = elevator.clone();
    let (obstruction_tx, mut obstruction_rx) = mpsc::channel(32);

    // Spawn a task to poll the obstruction switch state
    spawn(async move {
        data::obstruction(elevator_clone.clone(), obstruction_tx, Duration::from_millis(POLL_INTERVAL)).await;
    });

    // Spawn a task to update the obstruction state
    spawn(async move {
        while let Some(is_active) = obstruction_rx.recv().await {
            {
                let mut obstruction = obstruction_state_clone.write().await;
                *obstruction = is_active; // Update the obstruction switch state
            }
        }
    });
                                // 9 - READING: Obstruction switch (STOP)
============================================================================================
=======

                            // GET - NETWORK: Listen to manager (START)
============================================================================================
=======
    // We listen to manager hall delegation
    // Once we get a hall delegated to us
    // We save the order to a temp buffer
    // Then we save requests to hall UP/DOWN queue
    // This way if there are any more orders pending we only have to use shared resources only once
    //
    // In addition, if someone on the elevator pressed STOP button
    // This means we are in a emergency, witch in turn means we should stop listening to the outside world as well
    // The only thing that matters in a emergency situation is people inside the elevator cab
    // Because of this, in case of emergency stop, we also stop listening to the manager requests
    // We still receive manager data and keep track of whats going on in the network
    // However we simply disobey manager commands as this is an emergency
     // We also stop heartbeat, meaning manager will sooner or later realize something went wrong and divert requests to other elevators
    // Leaving our emergency stop elevator in piece until we have sorted stuff out
    //
    // In addition we will check all the manager requests, not just only ours
    // This way we can display all the active hall call through button LEDS later on in the process
    // We check this no matter the state, even in emergency state we update LEDs for hall
    // NOTE: In case of network disconnect, global LEDs will get set to 0

    // Create global hall LED display
    let global_leds_up: Arc<RwLock<HashSet<u8>>> = Arc::new(RwLock::new(HashSet::new()));
    let global_leds_down: Arc<RwLock<HashSet<u8>>> = Arc::new(RwLock::new(HashSet::new()));

    // Manager Hall Requests Thread ----------
    let manager_request_subscriber = network_session
        .declare_subscriber(MANAGER_TOPIC)
        .await
        .expect("Failed to declare subscriber for Manager Request Up");

    let state_clone = state.clone();
    let hall_up_queue_clone = hall_up_queue.clone();
    let hall_down_queue_clone = hall_down_queue.clone();
    spawn(async move {
        loop {
            match manager_request_subscriber.recv_async().await {
                Ok(message) => {
                    if let Some(elevator_requests) = parse_message_to_elevator_requests(message) {
                        // Check if the received data contains our elevator ID
                        if let Some(hall_requests) = elevator_requests.requests.get(&*ELEVATOR_NETWORK_ID.to_string()) {
                            // Use scoped locks to prevent holding lock for to long
                            let state = {
                                let state = state_clone.read().await;
                                state.clone() // Copy the state, avoiding unnecessary clones
                            };

                            // Finally ensure that our elevator is NOT in emergency (ie, no STOP button has been pressed)
                            // If elevator is in good state, we listen to the manager
                            // If elevator is in any emergency state, then we disobey manager orders by never reading them
                            if state != State::EmergencyStop && state != State::EmergencyStopIdle {
                                // Temporary buffers for up/down hall requests
                                let mut temp_up = HashSet::new();
                                let mut temp_down = HashSet::new();

                                // Iterate over our elevator's assigned hall requests
                                for (floor, hall) in hall_requests.iter().enumerate() {
                                    if hall[0] {
                                        temp_down.insert(floor as u8);
                                    }
                                    if hall[1] {
                                        temp_up.insert(floor as u8);
                                    }
                                }

                                // println!("DEBUG: Hall UP: {:#?}", temp_up);
                                // println!("DEBUG: Hall DOWN: {:#?}", temp_down);

                                // Efficiently update the shared HashSets in one go
                                // NOTE: We only update it if we read the difference between the current and received hall requests
                                // The READ lock comes in clutch by letting us read without sacrificing concurrency
                                // And enclosed in if statement we only use the actual lock in Write for a split second
                                // Combined with only writing when necessary this is super fast
                                // (The magic of rust compiler never cease to amaze me, WoooOOoowWw... *o*)
                                {
                                    let down_queue = {
                                        let down_queue = hall_down_queue_clone.read().await;
                                        down_queue.clone() // Clone the HashSet into a separate variable
                                    }; // Lock is released here

                                    if down_queue != temp_down {
                                        let mut down_queue = hall_down_queue_clone.write().await;
                                        *down_queue = temp_down;
                                    }
                                }
                                {
                                    let up_queue = {
                                        let up_queue = hall_up_queue_clone.read().await;
                                        up_queue.clone() // Clone the HashSet into a separate variable
                                    }; // Lock is released here

                                    if up_queue != temp_up {
                                        let mut up_queue = hall_up_queue_clone.write().await;
                                        *up_queue = temp_up;
                                    }
                                }
                            }
                        } else {
                            // println!("DEBUG: No hall requests found for Elevator ID: {}", *ELEVATOR_NETWORK_ID);
                        }
                    } else {
                        // println!("DEBUG: Received invalid data for Manager Request");
                    }
                }
                Err(e) => {
                    println!("Error receiving from topic: {}", MANAGER_TOPIC);
                    println!("Error code: {}", e);
                }
            }
        }
    });

    // Global LEDs from Manager Thread ----------
    let manager_request_subscriber = network_session
        .declare_subscriber(MANAGER_TOPIC)
        .await
        .expect("Failed to declare subscriber for Manager Request Up");

    let global_leds_up_clone = global_leds_up.clone();
    let global_leds_down_clone = global_leds_down.clone();
    spawn(async move {
        loop {
            match manager_request_subscriber.recv_async().await {
                Ok(message) => {
                    if let Some(elevator_requests) = parse_message_to_elevator_requests(message) {
                        // Save Global LED states
                        let mut global_up_temp = HashSet::new();
                        let mut global_down_temp = HashSet::new();

                        // Iterate through all received elevator hall requests
                        for (_elevator_id, hall_requests) in &elevator_requests.requests {
                            for (floor, hall) in hall_requests.iter().enumerate() {
                                if hall[0] {
                                    global_down_temp.insert(floor as u8);
                                }
                                if hall[1] {
                                    global_up_temp.insert(floor as u8);
                                }
                            }
                        }

                        // Efficiently update global LED hall requests
                        {
                            let current_global_led_down = {
                                let current_global_led_down = global_leds_down_clone.read().await;
                                current_global_led_down.clone() // Clone the HashSet into a separate variable
                            }; // Lock is released here

                            if current_global_led_down != global_down_temp {
                                let mut current_global_led_down = global_leds_down_clone.write().await;
                                *current_global_led_down = global_down_temp;
                            }
                        }
                        {
                            let current_global_led_up = {
                                let current_global_led_up = global_leds_up_clone.read().await;
                                current_global_led_up.clone() // Clone the HashSet into a separate variable
                            }; // Lock is released here

                            if current_global_led_up != global_up_temp {
                                let mut current_global_led_up = global_leds_up_clone.write().await;
                                *current_global_led_up = global_up_temp;
                            }
                        }
                    } else {
                        // println!("DEBUG: Received invalid data for Manager LEDs Request");
                    }
                }
                Err(e) => {
                    println!("Error receiving from topic: {}", MANAGER_TOPIC);
                    println!("Error code: {}", e);
                }
            }
        }
    });

    // Separate thread if we are outside network
    // If offline we must reset global LEDs
    let global_leds_up_clone = global_leds_up.clone();
    let global_leds_down_clone = global_leds_down.clone();
    let on_the_network_clone = on_the_network.clone();

    spawn(async move {
        let mut interval = time::interval(Duration::from_secs(2)); // Check every 2 seconds
```

```
        loop {
            interval.tick().await;

            // Check network state
            let network_status = *on_the_network_clone.read().await;
            if !network_status {
                println!("NETWORK DISCONNECTED - RESETTING GLOBAL LEDS!");

                // Reset global LEDs if offline
                *global_leds_up_clone.write().await = HashSet::new();
                *global_leds_down_clone.write().await = HashSet::new();
            }
        }
    });
    // GET - NETWORK: Listen to manager (STOP)
===============================================================================
=======

    // 2 - WRITING: Button order light (START)
===============================================================================
=======
    // PROBLEM:
    // - Updating button lights (CAB, UP, DOWN) involves toggling lights for all halls sequentially,
    //   but the elevator hardware IO is slow, causing high latency when toggling unnecessary lights.
    // - In a distributed network, hall requests can come from different elevators,
    //   meaning we need to track global requests as well as our own.
    //
    // SOLUTION:
    // - Use local HashSets (`local_cab_queue`, `local_hall_up_queue`, `local_hall_down_queue`) to track
    //   current light states and compare them with the combined real queues (`cab_queue`, `hall_up_queue`,
`hall_down_queue`).
    // - Merge global hall requests (`global_leds_up`, `global_leds_down`) with local ones before updating lights.
    // - Only toggle lights (ON/OFF) when a mismatch is detected:
    //   1. **Turn ON a light** if it's in either the local or global request queue but not already in the local LED state.
    //   2. **Turn OFF a light** if it's not in either queue but still exists in the local LED state.
    //
    // NETWORK FAILOVER HANDLING:
    // - If the network goes down or disconnects, the **global LED values reset to 0**, meaning all global hall request
LEDs turn OFF.
    // - However, **local hall requests remain ON** ensuring proper behavior in case of network failure.
    //
    // BENEFITS:
    // - Reduces unnecessary IO operations, minimizing latency.
    // - Ensures faster and consistent light updates.
    // - Scales efficiently with more halls or button types.
    // - Ensures hall lights remain on **even if the network fails**, preventing misleading visual indicators.

    // Control order button lights for cab calls and hall calls
    let cab_queue_clone = cab_queue.clone();
    let hall_up_queue_clone = hall_up_queue.clone();
    let hall_down_queue_clone = hall_down_queue.clone();
    let global_leds_up_clone = global_leds_up.clone();
let global_leds_down_clone = global_leds_down.clone();
let elevator_clone = elevator.clone();

spawn(async move {
    // Local HashMaps to keep track of the current button states
    let mut local_cab_queue: HashSet<u8> = HashSet::new();
    let mut local_hall_up_queue: HashSet<u8> = HashSet::new();
    let mut local_hall_down_queue: HashSet<u8> = HashSet::new();

    // Calculate perfect period so that we update all lights at predictable frequency
    let mut interval = time::interval(Duration::from_millis(POLL_INTERVAL));

    loop {
        // CAB Lights
        let cab_queue = cab_queue_clone.read().await;
        for hall in 0..*NUMBER_FLOORS {
            // Check if the light needs to be turned ON
            if cab_queue.contains(&hall) && !local_cab_queue.contains(&hall) {
                elevator_clone.call_button_light(hall, 2, true).await; // Turn ON CAB light
                local_cab_queue.insert(hall); // Update local state
            }
            // Check if the light needs to be turned OFF
            if !cab_queue.contains(&hall) && local_cab_queue.contains(&hall) {
                elevator_clone.call_button_light(hall, 2, false).await; // Turn OFF CAB light
                local_cab_queue.remove(&hall); // Update local state
            }
        }

        // Floor DOWN Lights
        let down_queue = hall_down_queue_clone.read().await;
        let global_down = global_leds_down_clone.read().await;

        // Create merged set of all active down requests (local + global)
        let merged_down: HashSet<u8> = down_queue.union(&*global_down).cloned().collect();

        for hall in 0..*NUMBER_FLOORS {
            if merged_down.contains(&hall) && !local_hall_down_queue.contains(&hall) {
                elevator_clone.call_button_light(hall, 1, true).await; // Turn ON DOWN light
                local_hall_down_queue.insert(hall);
            }
            if !merged_down.contains(&hall) && local_hall_down_queue.contains(&hall) {
                elevator_clone.call_button_light(hall, 1, false).await; // Turn OFF DOWN light
                local_hall_down_queue.remove(&hall);
            }
        }

        // Floor UP Lights
        let up_queue = hall_up_queue_clone.read().await;
        let global_up = global_leds_up_clone.read().await;

        // Create merged set of all active up requests (local + global)
        let merged_up: HashSet<u8> = up_queue.union(&*global_up).cloned().collect();
```

```
            for hall in 0..*NUMBER_FLOORS {
                if merged_up.contains(&hall) && !local_hall_up_queue.contains(&hall) {
                    elevator_clone.call_button_light(hall, 0, true).await; // Turn ON UP light
                    local_hall_up_queue.insert(hall);
                }
                if !merged_up.contains(&hall) && local_hall_up_queue.contains(&hall) {
                    elevator_clone.call_button_light(hall, 0, false).await; // Turn OFF UP light
                    local_hall_up_queue.remove(&hall);
                }
            }

            interval.tick().await;
        }
    });
                        // 2 - WRITING: Button order light (STOP)
=============================================================================================
=======

                        // 3 - WRITING: Floor indicator (START)
=============================================================================================
=======
    // Control hall indicator light
    let current_floor_light = Arc::clone(&current_floor);
    let elevator_clone = elevator.clone();
    spawn(async move {
        loop {
            let current_hall = {
                let current_hall = current_floor_light.read().await;
                *current_hall
            };

            if let Some(hall) = current_hall {
                elevator_clone.floor_indicator(hall).await;
            }

            tokio::time::sleep(Duration::from_millis(POLL_INTERVAL)).await; // Periodic update light
        }
    });
                        // 3 - WRITING: Floor indicator (STOP)
=============================================================================================
=======

                        // 5 - WRITING: Stop button light (START)
=============================================================================================
=======
    // Spawn a task to update the stop button light
    let stop_button_state_clone = Arc::clone(&stop_button_state);
    let elevator_clone = elevator.clone();
    spawn(async move {
        loop {
            let stop_button = {
```

```
            let stop_button = stop_button_state_clone.read().await;
            *stop_button
        };

        elevator_clone.stop_button_light(stop_button).await;

        tokio::time::sleep(Duration::from_millis(POLL_INTERVAL)).await; // Periodic update light
    }
});
// 5 - WRITING: Stop button light (STOP)
=========================================================================================================
=======

// SEND - NETWORK: Send heartbeat to manager (START)
=========================================================================================================
=========
// Send a steady heartbeat to show that this elevator node is in the network and is ready to receive requests
// NOTE: The only times we intentionally STOP sending heartbeat is in emergency state
// If someone on the cab presses STOP button we stop the heartbeat
// This way manager node and the rest of the network gets notified that something went wrong with our elevator
// This way some other elevator can handle our Hall calls
// Once we get back to normal states we resume the heartbeat
// Signaling to the network we are again ready to take the requests
let heartbeat_publisher = network_session
    .declare_publisher(TOPICS.heartbeat)
    .await
    .expect("Failed to declare heartbeat publisher");

let state_clone = state.clone();

spawn(async move {
    loop {
        // Use scoped locks to prevent holding lock for to long
        let state = {
            let state = state_clone.read().await;
            *state // Copy the state, avoiding unnecessary clones
        };

        // STOP sending heartbeat IF someone pressed the emergency STOP button
        if state != State::EmergencyStopIdle {
            heartbeat_publisher.put("BeepBoop ^-^".as_bytes()).await.expect("Failed to send heartbeat");
        }

        tokio::time::sleep(Duration::from_millis(HEARTBEAT_INTERVAL)).await;
    }
});
// SEND - NETWORK: Send heartbeat to manager (STOP)
=========================================================================================================
=========

// SEND - NETWORK: Backup Data (START)
=========================================================================================================
```

Innhald frå Rust-filer

```
=========
    // Create publishers for backing up data
    let backup_publisher = network_session
        .declare_publisher(TOPICS.backup_temp)
        .await
        .expect("Failed to declare Backup publisher");

    // Publish and backup data
    let state_clone = state.clone();
    let direction_clone = direction.clone();
    let current_floor_clone = current_floor.clone();
    let cab_queue_clone = cab_queue.clone();
    let hall_up_queue_clone = hall_up_queue.clone();
    let hall_down_queue_clone = hall_down_queue.clone();
    spawn(async move {
        loop {
            // Use scoped locks to prevent holding lock for to long
            let state = {
                let state = state_clone.read().await;
                state.clone()
            };
            let direction = {
                let direction = direction_clone.read().await;
                direction.clone()
            };
            let current_floor = {
                let current_floor = current_floor_clone.read().await;
                current_floor.clone()
            };
            let cab_queue = {
                let cab_queue = cab_queue_clone.read().await;
                cab_queue.clone()
            };
            let hall_up_queue = {
                let hall_up_queue = hall_up_queue_clone.read().await;
                hall_up_queue.clone()
            };
            let hall_down_queue = {
                let hall_down_queue = hall_down_queue_clone.read().await;
                hall_down_queue.clone()
            };

            // Format data to backup data format
            let backup_data = ElevatorBackup { state, direction, current_floor, cab_queue, hall_up_queue, hall_down_queue
};

            // Convert it into JSON format
            let json_backup_data = serde_json::to_string_pretty(&backup_data).expect("Failed to serialize backup data");

            // Send JSON to backup
            backup_publisher.put(json_backup_data.as_bytes()).await.expect("Failed to backup data");
            tokio::time::sleep(Duration::from_millis(BACKUP_INTERVAL)).await;
        }
    });
    // SEND - NETWORK: Backup Data (STOP)
============================================================================================
=========

    // SEND - NETWORK: Elevator States Update (START)
============================================================================================
=========
    // Data to send only when there is a change in any of the following states and this specific order:
    //
    // State: idle/moving/doorOpen
    // Floor: 0-255
    // Direction: Up/Down/Stop
    // Cab queue: [<floor 0: true/false>, .... , <floor N: true/false>]
    let elevator_states_publisher = network_session
        .declare_publisher(TOPICS.elevator_states)
        .await
        .expect("Failed to declare Elevator States publisher");

    // Request sending data
    let state_clone = state.clone();
    let current_floor_clone = current_floor.clone();
    let direction_clone = direction.clone();
    let cab_queue_clone = cab_queue.clone();

    // Since we only want to send data on changes, that means that we must keep track of all the changes internally
    // This way we know when there is a difference and if so we send the whole request until no changes
    spawn(async move {
        // Local copies to track changes
        let mut local_state = State::Idle;
        let mut local_floor = None::<u8>;
        let mut local_direction = Direction::Stop;
        let mut local_cab_queue: HashSet<u8> = HashSet::new();

        loop {
            // Use scoped locks to prevent holding multiple locks simultaneously
            let state = {
                let state = state_clone.read().await;
                state.clone() // Clone into a local variable, lock is released here
            };
            let floor = {
                let floor = current_floor_clone.read().await;
                floor.clone() // Clone into a local variable, lock is released here
            };
            let direction = {
                let direction = direction_clone.read().await;
                direction.clone() // Clone into a local variable, lock is released here
            };
            let cab_queue = {
                let cab = cab_queue_clone.read().await;
                cab.clone() // Clone into a local variable, lock is released here
            };

            // Check if any values have changed
            let state_changed = state != local_state;
            let floor_changed = floor != local_floor;
            let direction_changed = direction != local_direction;
            let cab_changed = cab_queue != local_cab_queue;

            // If any value changed, send an update
            if state_changed || floor_changed || direction_changed || cab_changed {
                // Format request into JSON ----------
                let formatted_state = match state {
                    State::Idle | State::EmergencyStop | State::EmergencyStopIdle => "idle".to_string(),
                    State::Up | State::Down => "moving".to_string(),
                    State::Door => "doorOpen".to_string(),
                };

                let formatted_floor = floor.unwrap_or(255); // If None, default to 255 (invalid floor)

                let formatted_direction = format!("{:?}", direction).to_lowercase(); // Convert direction enum to string

                let formatted_cab_queue: Vec<bool> = (0..*NUMBER_FLOORS).map(|floor|
cab_queue.contains(&floor)).collect();

                let elevator_states_data = ElevState {
                    behaviour: formatted_state,
                    floor: formatted_floor,
                    direction: formatted_direction,
                    cabRequests: formatted_cab_queue,
                };

                let elevator_states_data_formatted = serde_json::to_string(&elevator_states_data).expect("Failed to format
elevator states");

                // Send request ----------
                elevator_states_publisher
                    .put(elevator_states_data_formatted.as_bytes())
                    .await
                    .expect("Failed to send Elevator States");

                // Update local copies to prevent unnecessary updates
                local_state = state.clone();
                local_floor = floor.clone();
                local_direction = direction.clone();
                local_cab_queue = cab_queue.clone();
            }

            // Wait for the next interval, a small timeout to not overwhelm other threads
            tokio::time::sleep(Duration::from_millis(POLL_INTERVAL)).await;
        }
    });
```

Innhald frå Rust-filer

```
// SEND - NETWORK: Elevator States Update (STOP)
=============================================================================================
=========

// STATE MACHINE (START)
=============================================================================================
=======
    // Before starting state machine we wait a bit
     // This is because we want all values to be updated from sensors before we start running state machine for the
elevator
    tokio::time::sleep(Duration::from_millis(1000)).await;

    // Create publisher that sends remove hall button requests UP/DOWN to the manager and rest of the system
    let hall_requests_publisher = network_session
        .declare_publisher(HALL_REQUESTS_SYNC_TOPIC)
        .await
        .expect("Failed to declare Hall Requests publisher");

    // Declare all shared variables necessary for the state machine
    let state_clone = state.clone();
    let direction_clone = direction.clone();
    let cab_queue_clone = cab_queue.clone();
    let hall_up_queue_clone = hall_up_queue.clone();
    let hall_down_queue_clone = hall_down_queue.clone();
    let elevator_clone = elevator.clone();
    let current_floor_clone = current_floor.clone();
    let obstruction_state_clone = obstruction_state.clone();
    let stop_button_state_clone = stop_button_state.clone();

    spawn(async move {
        let mut motor_state = State::Idle;

        // Start with the data backed up state if it exists
        // We need to do some cursed way of ownership dereferencing
        // We move data to its own data variable independent of the lock
        // Its cursed, but thats what you get when dealing with custom data types that don't support this out the get go X-X
        let start_state = {
            let start_state = state_clone.read().await;
            (*start_state).clone() // Dereference and clone the value to make it independent
        };
        let mut _state = start_state.to_owned();
        let mut _prev_state = State::Idle;

        let start_direction = {
            let start_direction = direction_clone.read().await;
            (*start_direction).clone() // Dereference and clone the value to make it independent
        };
        let mut _direction = start_direction.to_owned();
        let mut _prev_direction = Direction::Stop;

        // Ensure the initial previous floor matches the current floor.
        // This prevents an edge case where, if the elevator crashes while between floors,
        // it forgets its last known state. Without this, the elevator could incorrectly
        // assume it should open its cab door upon restart, even if it is still moving.
        //
        // This issue only occurs if the same cab request is made again after a crash and
        // the elevator starts moving before completing its previous request.
        //
        // To prevent this, we set `visited_floor` to `current_floor`, ensuring that
        // any pending request to the last known floor is completed before shutting down.
        let mut visited_floor;
        {
            let current_floor = current_floor_clone.read().await;
            visited_floor = current_floor.unwrap_or(0); // Default to 0 if None
        }

        // Start elevator state machine at 1st hall
        // This forces elevator to go down on startup
        // Both for convenience and for safety
        {
            // Acquire a lock on the cab_calls_clone Mutex
            let mut cab_queue = cab_queue_clone.write().await;

            // Insert 0 to start elevator at 0th hall
            cab_queue.insert(0);
        }

        loop {
            // Save latest state from state machine so that backing up thread can handle backing up state
            // Lets be hones here... this function is cursed X-X
            // But cloning the lock is insanely slow, and stunts the whole thread
            // Its because we use custom enum state that does not natively support copying, meaning we have to clone
            // Cloning data is slow
            // This is the best I could think of for bypassing cloning :/
            // The worst part is that this actually works, it solves the issue and the code stays fast *O*
            //
             // "Do you think God stays in heaven because he, too, lives in fear of what he's created here on earth?" - Robert Rodriguez, writer of Spy Kids 2
            {
                if _prev_state != _state {
                    let mut state_backup = state_clone.write().await;

                    match _state {
                        State::Idle => {
                            *state_backup = State::Idle;
                            _prev_state = State::Idle;
                        }
                        State::Up => {
                            *state_backup = State::Up;
                            _prev_state = State::Up;
                        }
                        State::Down => {
                            *state_backup = State::Down;
                            _prev_state = State::Down;
                        }
                        State::Door => {
                            *state_backup = State::Door;
                            _prev_state = State::Door;
                        }
                        State::EmergencyStop => {
                            *state_backup = State::EmergencyStop;
                            _prev_state = State::EmergencyStop;
                        }
                        State::EmergencyStopIdle => {
                            *state_backup = State::EmergencyStopIdle;
                            _prev_state = State::EmergencyStopIdle;
                        }
                    }
                }
            } // Lock is released here

            // Save direction into shared variable
            // This way if its updated elevator request thread will send a request to the manager with updated direction state
            {
                if _prev_direction != _direction {
                    _prev_direction = _direction;

                    let mut direction_shared = direction_clone.write().await;
                    *direction_shared = match _direction {
                        Direction::Up => Direction::Up,
                        Direction::Stop => Direction::Stop,
                        Direction::Down => Direction::Down,
                    }
                }
            } // Lock is released here

            // Get stop button state
            let stop_button = {
                let stop_button = stop_button_state_clone.read().await;
                *stop_button
            }; // Lock is released here

            // Get current hall we are on
            let current_floor = current_floor_clone.read().await.unwrap_or_else(|| {
                println!();
                println!("#============================================================#");
                println!("ERROR: Current floor is not set! Exiting program.");
                println!("ERROR: Check that elevator IO is connected for floor sensor");
                println!("#============================================================#");
                println!();
                std::process::exit(1);
            }); // Lock is released here

            // Check if we have any cab calls under way
            // Clone the current state of cab_que into a separate variable
            // This way the lock is used up immediately and frees up the resource for other threads much faster
```

```
        let cab_queue = {
            let cab_queue = cab_queue_clone.read().await;
            cab_queue.clone() // Clone the HashSet into a separate variable
        }; // Lock is released here

        // Check if we have any halls calls under way
           // Clone the current state of hall calls into a separate variable so that other threads can use same resources
faster
        let hall_up_queue = {
            let hall_up_queue = hall_up_queue_clone.read().await;
            hall_up_queue.clone() // Clone the HashSet into a separate variable
        }; // Lock is released here
        let hall_down_queue = {
            let hall_down_queue = hall_down_queue_clone.read().await;
            hall_down_queue.clone() // Clone the HashSet into a separate variable
        }; // Lock is released here

        match _state {
            State::Idle => {
                let mut found_solution = None;

                // Check for emergency stop first
                found_solution = found_solution.or_else(|| state_machine::handle_emergency_stop(stop_button));

                // Handle cab requests on current floor while we are still not moving
                 found_solution = found_solution.or_else(|| state_machine::handle_cab_request_current_floor(&cab_queue,
current_floor));

                // Handle direction-specific logic
                match _direction {
                    Direction::Stop => {
                        // Handle hall calls if we have exhausted all requests
                        // Always try to find hall requests up first, only then down requests
                        found_solution = found_solution.or_else(||
state_machine::handle_hall_up_request_current_floor(&hall_up_queue, current_floor));
                        found_solution = found_solution.or_else(||
state_machine::handle_hall_down_request_current_floor(&hall_down_queue, current_floor));

                        // Handle random direction logic if all other options were exhausted
                                found_solution = found_solution.or_else(|| state_machine::find_random_request(&cab_queue,
&hall_up_queue, &hall_down_queue, current_floor));
                    }
                    Direction::Up => {
                        // Before moving check that there are no new UP requests from the same floor
                        found_solution = found_solution.or_else(||
state_machine::handle_hall_up_request_current_floor(&hall_up_queue, current_floor));

                        // Look for requests above the current hall
                                 found_solution = found_solution.or_else(|| state_machine::find_request_above(&cab_queue,
current_floor));
                              found_solution = found_solution.or_else(|| state_machine::find_request_above(&hall_up_queue,
current_floor));
```

```
                        // Look for any requests above that are DOWN if no more up queues that way
                            found_solution = found_solution.or_else(|| state_machine::find_request_above(&hall_down_queue,
current_floor));

                        // If no further upwards requests found
                        // Search for special case requests that go opposite to the normal direction (ie DOWN)
                            found_solution = found_solution.or_else(|| state_machine::find_request_below(&cab_queue,
current_floor));
                            found_solution = found_solution.or_else(|| state_machine::find_request_below(&hall_down_queue,
current_floor));
                    }
                    Direction::Down => {
                        // Before moving check that there are no new DOWN requests from the same floor
                            found_solution = found_solution.or_else(||
state_machine::handle_hall_down_request_current_floor(&hall_down_queue, current_floor));

                        // Look for requests below the current hall
                            found_solution = found_solution.or_else(|| state_machine::find_request_below(&cab_queue,
current_floor));
                            found_solution = found_solution.or_else(|| state_machine::find_request_below(&hall_down_queue,
current_floor));

                        // Look for any requests bellow that are UP if no more down queues that way
                            found_solution = found_solution.or_else(|| state_machine::find_request_below(&hall_up_queue,
current_floor));

                        // If no further downwards requests found
                        // Search for special case requests that go opposite to the normal direction (ie UP)
                            found_solution = found_solution.or_else(|| state_machine::find_request_above(&cab_queue,
current_floor));
                            found_solution = found_solution.or_else(|| state_machine::find_request_above(&hall_up_queue,
current_floor));
                    }
                }

                // Update state if a solution was found
                if let Some(new_state) = found_solution {
                    _state = new_state;

                    // Special case exceptions to the rule
                    // If no new requests in the previous direction
                    // However there is a request from the opposite direction
                    // We must handle that request this turn
                    // So that means we must also clear the special case request light as well
                    if _direction == Direction::Down && _state == State::Up {
                        // Clear the Floor UP signal
                        {
                            let mut hall_up_queue = hall_up_queue_clone.write().await;
                            hall_up_queue.remove(&current_floor);

                            let request = create_hall_request_json(
                                None,
                                None,
                                Some(current_floor), // UP Remove
                                None,
                            );

                            hall_requests_publisher.put(request.as_bytes()).await.expect("Failed to publish hall button UP");
                        }
                    } else if _direction == Direction::Up && _state == State::Down {
                        // Clear the Floor DOWN signal
                        {
                            let mut hall_down_queue = hall_down_queue_clone.write().await;
                            hall_down_queue.remove(&current_floor);

                            let request = create_hall_request_json(
                                None,
                                None,
                                None,
                                Some(current_floor), // DOWN Remove
                            );

                            hall_requests_publisher.put(request.as_bytes()).await.expect("Failed to publish hall button UP");
                        }
                    }
                } else {
                    // No solution was found
                    // Reset direction
                    _direction = Direction::Stop;

                    // Set motor to IDLE
                    if motor_state != State::Idle {
                        elevator_clone.motor_direction(0).await;

                        motor_state = State::Idle;
                    }
                }
            }
            State::Up => {
                // Set motor UP
                if motor_state != State::Up {
                    elevator_clone.motor_direction(1).await;

                    motor_state = State::Up;
                }

                _direction = Direction::Up;

                // If we hit a different floor go into Idle state
                // These it will handle the rest of the logic
                if visited_floor != current_floor {
                    _state = State::Idle;
                }
                // Check for emergency stop button
                // Check for it last to ensure it overwrites any other state set in case of emergency
                if stop_button {
                    _state = State::EmergencyStop;
                }

                // Update visited floor to the latest floor we are at
                // No matter if we found a solution or not
                // This is the floor that we have now visited
                // no more queues will be handled for this visited floor at this stage
                // Even if they come in to late, new queues for this floor will have to wait
                visited_floor = current_floor;

                //println!("DEBUG: Current Floor {:#?}", current_floor);
            }
            State::Down => {
                // Set motor DOWN
                if motor_state != State::Down {
                    elevator_clone.motor_direction(255).await;

                    motor_state = State::Down;
                }

                _direction = Direction::Down;

                // If we hit a different floor go into Idle state
                // These it will handle the rest of the logic
                if visited_floor != current_floor {
                    _state = State::Idle;
                }

                // Check for emergency stop button
                // Check for it last to ensure it overwrites any other state set in case of emergency
                if stop_button {
                    _state = State::EmergencyStop;
                }

                // Update visited floor to the latest floor we are at
                // No matter if we found a solution or not
                // This is the floor that we have now visited
                // no more queues will be handled for this visited floor at this stage
                // Even if they come in to late, new queues for this floor will have to wait
                visited_floor = current_floor;

                //println!("DEBUG: Current Floor {:#?}", current_floor);
            }
            State::Door => {
                // Set motor IDLE
                if motor_state != State::Idle {
                    elevator_clone.motor_direction(0).await;
```

```
                        motor_state = State::Idle;
                    }

                    // remove all request on this specific hall we stopped at
                    // Exception: Don't remove the requests at the opposite site we were going at, as per specification :)
                     // NOTE: We also send the remove request to the network so that everyone on the network knows that we
have handled the request and should remove it from the requests
                    {
                        let mut cab_queue = cab_queue_clone.write().await;
                        cab_queue.remove(&current_floor);
                    }
                    {
                        match _direction {
                            Direction::Down => {
                                if current_floor == 0 {
                                    {
                                        let mut hall_up_queue = hall_up_queue_clone.write().await;
                                        hall_up_queue.remove(&current_floor);
                                    } // UP Remove
                                    {
                                        let mut hall_down_queue = hall_down_queue_clone.write().await;
                                        hall_down_queue.remove(&current_floor);
                                    } // DOWN Remove

                                    let request = create_hall_request_json(
                                        None,
                                        None,
                                        Some(current_floor), // UP Remove
                                        Some(current_floor), // DOWN Remove
                                    );

                                    hall_requests_publisher.put(request.as_bytes()).await.expect("Failed to publish hall button UP");
                                } else {
                                    {
                                        let mut hall_down_queue = hall_down_queue_clone.write().await;
                                        hall_down_queue.remove(&current_floor);
                                    } // DOWN Remove

                                    let request = create_hall_request_json(
                                        None,
                                        None,
                                        None,
                                        Some(current_floor), // DOWN Remove
                                    );

                                    hall_requests_publisher.put(request.as_bytes()).await.expect("Failed to publish hall button UP");
                                }
                            }
                            Direction::Up => {
                                if current_floor == (*NUMBER_FLOORS - 1) {
                                    {
                                        let mut hall_up_queue = hall_up_queue_clone.write().await;
                    hall_up_queue.remove(&current_floor);
                } // UP Remove
                {
                    let mut hall_down_queue = hall_down_queue_clone.write().await;
                    hall_down_queue.remove(&current_floor);
                } // DOWN Remove

                let request = create_hall_request_json(
                    None,
                    None,
                    Some(current_floor), // UP Remove
                    Some(current_floor), // DOWN Remove
                );

                hall_requests_publisher.put(request.as_bytes()).await.expect("Failed to publish hall button UP");
            } else {
                {
                    let mut hall_up_queue = hall_up_queue_clone.write().await;
                    hall_up_queue.remove(&current_floor);
                } // UP Remove

                let request = create_hall_request_json(
                    None,
                    None,
                    Some(current_floor), // UP Remove
                    None,
                );

                hall_requests_publisher.put(request.as_bytes()).await.expect("Failed to publish hall button UP");
            }
        }
        _ => {
            {
                let mut hall_up_queue = hall_up_queue_clone.write().await;
                hall_up_queue.remove(&current_floor);
            } // UP Remove
            {
                let mut hall_down_queue = hall_down_queue_clone.write().await;
                hall_down_queue.remove(&current_floor);
            } // DOWN Remove

            let request = create_hall_request_json(
                None,
                None,
                Some(current_floor), // UP Remove
                Some(current_floor), // DOWN Remove
            );

            hall_requests_publisher.put(request.as_bytes()).await.expect("Failed to publish hall button UP");
        }
    }
}
```

```
                    // Open the door
                    elevator_clone.door_light(true).await;

                    // Timeout to wait for people to get out/in
                    tokio::time::sleep(Duration::from_millis(3000)).await;

                    // Check obstructions
                    while *obstruction_state_clone.read().await {
                        // Do nothing while we are blocked
                    }

                    // Close the door
                    elevator_clone.door_light(false).await;

                    _state = State::Idle;
                }
                State::EmergencyStop => {
                    // If user has pressed stop button we think of it as emergency
                    // Stop the elevator immediately
                    // Clear all cab calls
                    // Clear all hall calls
                    // Go into stop idle state
                    if motor_state != State::Idle {
                        elevator_clone.motor_direction(0).await;

                        motor_state = State::Idle;
                    }

                    {
                        let mut cab_queue = cab_queue_clone.write().await;
                        cab_queue.clear();
                    }

                    // NOTE: We don't signal to the rest of the network that we have removed our requests (ie publish remove requests)
                    // The network itself will figure out something went wrong with the elevator since we don't handle our requests no longer
                    // We will also STOP publishing heartbeat, prompting manager response to our unhandled requests
                    // This in turn will prompt the manager node to reallocate requests where it needs to be after noticing this elevator is in emergency state
                    {
                        let mut hall_up_queue = hall_up_queue_clone.write().await;
                        hall_up_queue.clear();
                    }

                    {
                        let mut hall_down_queue = hall_down_queue_clone.write().await;
                        hall_down_queue.clear();
                    }

                    _state = State::EmergencyStopIdle;
```

```
}
State::EmergencyStopIdle => {
    // Check if the previous direction was nothing, indicating we were idling
    // If so just go back to idling
    if _direction == Direction::Stop {
        _state = State::Idle;
    }

    // If it wasn't idle state we were in before, that means we are in between floors
    // We have to manage this a bit more carefully
    // Check if something new has happened in the CAB
    // NOTE: We ignore the outside hall requests and the world as we are in an emergency
    let something_new = !cab_queue.is_empty();

    // We wait in stop state until something new happens
    if something_new == true {
        // Something new happened
        // We need to check witch state we should go to

        // Sometimes elevator gets stopped between floors
        // check what the previous direction of movement was
        // If there is a cab_queue, we check if the next floor is out previous floor
        // If so we need to go to that floor before going to Idle
        // Otherwise its a different floor so Idle state can handle logic of it for us
        // NOTE: Again, we only care about what is happening inside the cab because we are in emergency
        // This means we don't care about the outside world

        // Check if the previous floor we departed before we stopped is in request queue
        let request_to_same_floor = cab_queue.contains(&current_floor);

        // If the request of the same previous floor is not there we are good
        // We can go to Idle state that will take case of things for us
        if !request_to_same_floor {
            // We should go down to the
            _state = State::Idle;
        } else {
            // Since the floor we want to go to now is the same as before we do nothing
            // This is because we tried to get it to work to go back to the floor
            // However for this we need to set elevator state HARDWARE wise to different floor
            // This way it thinks its on different floor and we can go to our floor
            // However microcontroller/Arduino saves the last state it had, and you can't edit it
            // Even if you try sending reloading the config, it will still fail
            // The only way to say to hardware that they need to clear their state is to turn the power off
            // And because of this it is not realizable to go back to the same floor T_T
            // So instead, if the button of that choice is clicked, we just delete it from the queue

            {
                let mut cab_queue = cab_queue_clone.write().await;
                cab_queue.remove(&current_floor);
            }
        }
    } else {
```

```
                        // Stay stuck in stop state
                    }
                }
            }
        }
    });
                                                                                    //                    STATE                    MACHINE                    (STOP)
============================================================================================
=======

    loop {
        yield_now().await;
    }
}
```

Innhald frå Rust-filer

Fil: 3701d923/TTK4145_Real_Time_Programming-main/src/elevator_process_pair.rs

```
// * NOTE: We don't actually need to run the node in "sudo", however for consistency with other nodes that require "sudo" we have made this node also run in "sudo"
// * NOTE: Every new elevator needs its own unique ELEVATOR_NETWORK_ID and ELEVATOR_HARDWARE_PORT
// *
// * To run this process:
// * $ sudo -E ELEVATOR_NETWORK_ID=<ID> ELEVATOR_HARDWARE_PORT=<PORT> NUMBER_FLOORS=<NUMBER FLOORS> cargo run --bin elevator_process_pair

use std::env;
use std::io::Write;
use std::net::TcpStream;
use std::process::{Command, ExitStatus};
use std::thread;
use std::time::Duration; // For write_all

/// Start the elevator process with the given network ID and hardware port using sudo.
fn start_elevator(network_id: &str, hardware_port: &str, number_floors: &str) -> ExitStatus {
    println!(
        "Starting elevator process with \n
        ELEVATOR_NETWORK_ID={} \n
        ELEVATOR_HARDWARE_PORT={} \n
        NUMBER_FLOORS={} \n",
        network_id, hardware_port, number_floors
    );

    Command::new("sudo")
        .arg("-E") // Preserve environment variables
        .env("ELEVATOR_NETWORK_ID", network_id)
        .env("ELEVATOR_HARDWARE_PORT", hardware_port)
        .env("NUMBER_FLOORS", number_floors)
        .arg("cargo")
        .arg("run")
        .arg("--bin")
        .arg("elevator") // Specify the elevator binary
        .status()
        .expect("Failed to start elevator process")
}

/// Connect to the elevator hardware to send the stop motor command.
fn stop_elevator_motor(hardware_port: &str) {
    let address = format!("localhost:{}", hardware_port);
    match TcpStream::connect(address) {
        Ok(mut socket) => {
            let buf = [1, 0, 0, 0]; // Command to stop the motor.
            if let Err(err) = socket.write_all(&buf) {
                eprintln!("Failed to send stop motor command: {}", err);
            }
        }
        Err(err) => {
            eprintln!("Failed to connect to the elevator system: {}", err);
        }
    }
}

fn main() {
    // Retrieve the environment variables.
    let network_id = env::var("ELEVATOR_NETWORK_ID").unwrap_or_else(|_| "0".to_string());
    let hardware_port = env::var("ELEVATOR_HARDWARE_PORT").unwrap_or_else(|_| "15657".to_string());
    let number_floors = env::var("NUMBER_FLOORS").unwrap_or_else(|_| "4".to_string());

    loop {
        // Start the elevator process.
        let status = start_elevator(&network_id, &hardware_port, &number_floors);

        // For safety, ensure the elevator motor is stopped.
        println!("Ensuring elevator motor is stopped...");
        stop_elevator_motor(&hardware_port);

        // Monitor process exit status.
        if let Some(code) = status.code() {
            println!("Elevator process exited with code: {}", code);
            if code == 0 {
                println!("Elevator process exited successfully. Exiting monitor.");
                break;
            }
        } else {
            println!("Elevator process was terminated by a signal.");
        }

        // Delay before restarting.
        println!("Restarting elevator process in 5 seconds...");
        thread::sleep(Duration::from_secs(5));
    }
}
```

Innhald frå Rust-filer

Fil: 3701d923/TTK4145_Real_Time_Programming-main/src/manager.rs

```
// * NOTE: Some networking libraries requiring root privileges (for eks for pinging to router to read if we are still on the same network)
// * NOTE: Every new manager needs its own unique MANAGER_ID
// *
// * Because of these factors we must run this process as follows:
// * $ sudo -E MANAGER_ID=<ID> ELEVATOR_NETWORK_ID_LIST="[<ID 1>,<ID 2>,...,<ID N>]" cargo run --bin manager

// Library that allows us to use environment variables or command-line arguments to pass variables from terminal to the program directly
use std::env;

// Libraries for multithreading in cooperative mode
use std::sync::Arc;
use tokio::sync::RwLock; // For optimization RwLock for data that is read more than written to. // Mutex for 1-1 ratio of read write to
use tokio::time::{sleep, Duration};

// Libraries for highly customizable distributed network mode setup
use std::fs;
use std::path::Path;
use zenoh::Config;

// Libraries for distributed network
use zenoh::open;

// Libraries for network status and diagnostics
// * NOTE: Because of some functions in utils library that uses ICMP sockets, witch are ONLY available for root user, we must run our program as sudo cargo run
use elevator_system::distributed_systems::utils::{get_router_ip, network_status, parse_message_to_string, wait_with_timeout, NetworkStatus};

// Libraries for constructing data structures that are thread safe
use once_cell::sync::Lazy;

// Library for the cost function algorithm
use elevator_system::elevator_algorithm::cost_algorithm::run_cost_algorithm;
use elevator_system::elevator_algorithm::utils::AlgoInput;

// Global Variables ----------
const SYNC_INTERVAL: u64 = 1000; // ms
const NETWORK_CHECK_INTERVAL: u64 = 5000; // ms
const HEARTBEAT_INTERVAL: u64 = 1000; // ms (*Taken from elevator.rs node)

const LEADER_TOPIC: &str = "sync/manager/leader";

const ELEVATOR_DATA_SYNC_TOPIC: &str = "sync/elevator/data/synchronized";
const MANAGER_TOPIC: &str = "temp/manager/request";

// Set up environment variables ----------
```

Innhald frå Rust-filer

```
// Get the MANAGER_ID from the environment variable, defaulting to 0 if not set
// !NOTE: Every new Rust process needs their own unique MANAGER_ID
static MANAGER_ID: Lazy<i64> = Lazy::new(|| {
    env::var("MANAGER_ID")
        .unwrap_or_else(|_| "0".to_string())
        .parse()
        .expect("MANAGER_ID must be a valid integer")
});

// Get the ELEVATOR_NETWORK_ID_LIST from the environment variable
// Defaulting to [0] if not set
static ELEVATOR_NETWORK_ID_LIST: Lazy<Vec<i64>> = Lazy::new(|| {
    // Expect the variable in the form "[1,2,3]"
    let list_str = env::var("ELEVATOR_NETWORK_ID_LIST").unwrap_or_else(|_| "[0]".to_string());
    list_str
        .trim_matches(|c| c == '[' || c == ']')
        .split(',')
        .map(|s| s.trim().parse().expect("Invalid elevator id in list"))
        .collect()
});

// Distributed Network Topics of interest ----------
static DATA_STREAMS_ELEVATOR_HEARTBEATS: Lazy<Vec<String>> = Lazy::new(||
ELEVATOR_NETWORK_ID_LIST.iter().map(|&id| format!("stor/elevator{}/heartbeat", id)).collect());

#[tokio::main]
async fn main() {
                                // Distributed Network Initialization (START)
===============================================================================
=========
    println!("MANAGER_ID: {}", *MANAGER_ID);
    println!("ELEVATOR_NETWORK_ID_LIST: {:#?}", *ELEVATOR_NETWORK_ID_LIST);

    // Specify path to highly customable network modes for distributed networks
    // Most important settings: peer-2-peer and scouting to alow multicast and robust network connectivity
    // Then Load configuration from JSON5 file
    // Finally initialize networking session
    let networking_config_path = Path::new("network_config.json5");

    let networking_config_data = fs::read_to_string(networking_config_path).expect("Failed to read the
network_config.json5 file");
    let config: Config = Config::from_json5(&networking_config_data).expect("Failed to parse the network_config.json5
file");

    let network_session = open(config).await.expect("Failed to open Zenoh session");
                                // Distributed Network Initialization (STOP)
===============================================================================
=========

                                          // Network Monitoring (START)
===============================================================================
=========
    // Spawn a separate task to check network status every so often
    // If we detect we have been disconnected from the network we kill ourselves
    tokio::spawn(async move {
        let router_ip = match get_router_ip().await {
            Some(ip) => ip,
            None => {
                println!("#========================================#");
                println!("ERROR: Failed to retrieve the router IP");
                println!("Killing myself...");
                println!("Gugu gaga *O*");
                println!("#========================================#");
                std::process::exit(1);
            }
        };

        loop {
            match network_status(router_ip).await {
                NetworkStatus::Connected => {
                    // Do nothing
                }
                NetworkStatus::Disconnected => {
                    println!("#========================================#");
                    println!("ERROR: Disconnected from the network!");
                    println!("Killing myself...");
                    println!("Shiding and crying T_T");
                    println!("#========================================#");
                    std::process::exit(1);
                }
            }

            sleep(Duration::from_millis(NETWORK_CHECK_INTERVAL)).await;
        }
    });
                                        // Network Monitoring (STOP)
========================================================================================
=========

                                        // Synchronization (START)
========================================================================================
=========
      let leader_publisher = network_session.declare_publisher(LEADER_TOPIC).await.expect("Failed to declare leader
publisher");

    let leader_subscriber = network_session.declare_subscriber(LEADER_TOPIC).await.expect("Failed to declare leader
subscriber");

    let leader = Arc::new(RwLock::new(false));

    // Leader monitoring task ----------
    {
        let leader = leader.clone();
        let leader_elect_interval = SYNC_INTERVAL * 5; // 5x sync because we want to make sure everyone who wants to
be a leader has broadcasted it at least once

        tokio::spawn(async move {
            loop {
                // Wait for a leader broadcast within the election interval
                let result = wait_with_timeout(leader_elect_interval, leader_subscriber.recv_async()).await;

                // Check the results
                // If we got a time-out, that means no one else on the network wants to be a leader
                // => become default leader
                // If there is someone else trying to become the leader
                // => Chose leader with lowest ID
                if let Some(message) = result {
                    // Parse leader ID from the announcement
                    let id = parse_message_to_string(message);

                    if let Ok(leader_id) = id.parse::<i64>() {
                        let mut are_we_leader = leader.write().await;

                        if leader_id < *MANAGER_ID {
                            *are_we_leader = false; // Step down from leadership
                        } else {
                            *are_we_leader = true; // Become leader
                        }
                    }
                } else {
                    // No leader broadcast received within the timeout
                    let mut is_leader_lock = leader.write().await;
                    *is_leader_lock = true; // Default to becoming the leader
                }
            }
        });
    }

    // Leader broadcasting task ----------
    {
        let leader = leader.clone();
        let leader_broadcast_interval = SYNC_INTERVAL;

        tokio::spawn(async move {
            loop {
                if *leader.read().await {
                    leader_publisher
                        .put((*MANAGER_ID).to_string().as_bytes())
                        .await
                        .expect("Failed to announce leadership");
                }

                sleep(Duration::from_millis(leader_broadcast_interval)).await;
            }
        });
    }
```

Innhald frå Rust-filer

```
                                                // Synchronization (STOP)
===========================================================================================
=========

                                    // Elevator Heartbeat Monitoring (START)
===========================================================================================
=========
    // Listen to each heartbeat
    // If heartbeat stopped after a while updated shared resource
    // Once it starts up again it will update shared resource again
    let elevators_alive = Arc::new(RwLock::new(vec![true; ELEVATOR_NETWORK_ID_LIST.len()])); // Assume all
elevators start alive

    // Loop through the whole Elevator Heartbeat list
    // Each elevator gets its own dedicated thread for listening at its heartbeat
    // If any anomalies or to timeout occurs, assume elevator dead
    // Otherwise keep holding the elevator alive
    for elevator_heartbeat_index in 0..ELEVATOR_NETWORK_ID_LIST.len() {
        // Set up resources for the local elevator thread
        let topic = DATA_STREAMS_ELEVATOR_HEARTBEATS
            .get(elevator_heartbeat_index)
            .expect("Invalid heartbeat index")
            .clone(); // Clone to avoid moving the String

        let elevator_heartbeat_subscriber = network_session
            .declare_subscriber(&topic) // Use &topic to avoid moving the String
            .await
            .expect("Failed to declare Elevator Heartbeat subscriber");

        let elevators_alive_clone = elevators_alive.clone();

        let heartbeat_dead_interval = HEARTBEAT_INTERVAL * 5; // 5x heartbeat interval because we want to make sure
everyone who wants to be a heartbeat has broadcasted it at least once

        tokio::spawn(async move {
            loop {
                // Wait for a leader broadcast within the election interval
                let result = wait_with_timeout(heartbeat_dead_interval, elevator_heartbeat_subscriber.recv_async()).await;

                // Check the results
                if let Some(message) = result {
                    let parsed_message = parse_message_to_string(message); // Convert Zenoh message to string

                    if !parsed_message.trim().is_empty() {
                        // Message is valid (not empty and not NaN)
                        // Elevator is alive
                        {
                            let mut elevators_alive = elevators_alive_clone.write().await;
                            elevators_alive[elevator_heartbeat_index] = true;
                        }
                    } else {
                        {
                            let mut elevators_alive = elevators_alive_clone.write().await;
                            elevators_alive[elevator_heartbeat_index] = false;
                        }
                    }
                } else {
                    // No heartbeat
                    {
                        let mut elevators_alive = elevators_alive_clone.write().await;
                        elevators_alive[elevator_heartbeat_index] = false;
                    }
                }
            }
        });
    }
                                                    // Elevator Heartbeat Monitoring (STOP)
============================================================================================
=========

                                                                          // Manager (START)
============================================================================================
=========
    let elevator_data_sync_subscriber = network_session
        .declare_subscriber(ELEVATOR_DATA_SYNC_TOPIC)
        .await
        .expect("Failed to declare Elevator Data Synchronization subscriber");

        let manager_publisher = network_session.declare_publisher(MANAGER_TOPIC).await.expect("Failed to declare
Manager publisher");

    let leader_clone = leader.clone();
    let elevators_alive_clone = elevators_alive.clone();

    tokio::spawn(async move {
        // Wait for new messages
        while let Ok(message) = elevator_data_sync_subscriber.recv_async().await {
            // NOTE: Only run cost function if we are the leader
            // Otherwise, just wait for new messages
            if *leader_clone.read().await {
                let json_str = parse_message_to_string(message);

                // Parse JSON into a struct
                        let mut parsed_data: AlgoInput = serde_json::from_str(&json_str).expect("Failed to parse elevator state
JSON");

                // Filter out dead elevators based on `elevators_alive` index
                let elevators_alive = elevators_alive_clone.read().await;
                for (index, &is_alive) in elevators_alive.iter().enumerate() {
                    if !is_alive {
                        if let Some(elevator_id) = ELEVATOR_NETWORK_ID_LIST.get(index) {
                            parsed_data.states.remove(&elevator_id.to_string());
                        }
                    }
                }

                // println!("DEBUGGING: Alive?: {:#?}", elevators_alive);

                // Serialize updated JSON
                let filtered_json = serde_json::to_string(&parsed_data).expect("Failed to reserialize JSON");

                // println!("DEBUGGING: Input: {:#?}", filtered_json);

                // Run cost function with filtered JSON
                let result = run_cost_algorithm(filtered_json).await;

                // println!("DEBUGGING: Output: {:#?}", result);

                // Publish the result
                manager_publisher.put(result.as_bytes()).await.expect("Failed to publish result");
            }
        }
    });

                                                            //              Manager              (STOP)
===========================================================================================
=========

    // Keep the program running
    loop {
        tokio::task::yield_now().await; // Yield to other tasks
    }
}
```

Innhald frå Rust-filer

Fil: 3701d923/TTK4145_Real_Time_Programming-main/src/database_process_pair.rs

```
// * NOTE: Some networking libraries requiring root privileges (for eks for pinging to router to read if we are still on the same network)
// * NOTE: Every new database needs its own unique DATABASE_ID
// *
// * Because of these factors we must run this process as follows:
// * $ 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

use std::env;
use std::process::{Command, ExitStatus};
use std::thread;
use std::time::Duration;

fn start_database(database_network_id: &str, elevator_network_id_list: &str, number_floors: &str) -> ExitStatus {
    println!(
        "Starting database process with \n
        DATABASE_NETWORK_ID={} \n
        ELEVATOR_NETWORK_ID_LIST={} \n
        NUMBER_FLOORS={} \n",
        database_network_id, elevator_network_id_list, number_floors
    );

    Command::new("sudo")
        .arg("-E") // Preserve environment
        .env("DATABASE_NETWORK_ID", database_network_id)
        .env("ELEVATOR_NETWORK_ID_LIST", elevator_network_id_list)
        .env("NUMBER_FLOORS", number_floors)
        .arg("cargo")
        .arg("run")
        .arg("--bin")
        .arg("database") // Specify the database binary
        .status()
        .expect("Failed to start database process")
}

fn main() {
    // Ensure DATABASE_NETWORK_ID is passed to the parent process
        let database_network_id = env::var("DATABASE_NETWORK_ID").expect("DATABASE_NETWORK_ID must be set");
    // Retrieve ELEVATOR_NETWORK_ID_LIST (defaulting to "[0]" if not set)
    let elevator_network_id_list = env::var("ELEVATOR_NETWORK_ID_LIST").unwrap_or_else(|_| "[0]".to_string());
    // Retrieve NUMBER_FLOORS (defaulting to "4" if not set)
    let number_floors = env::var("NUMBER_FLOORS").unwrap_or_else(|_| "4".to_string());

    loop {
        let status = start_database(&database_network_id, &elevator_network_id_list, &number_floors);

        if let Some(code) = status.code() {
            println!("Database process exited with code: {}", code);
            if code == 0 {
                println!("Database process exited successfully. Exiting monitor.");
                break;
            }
        } else {
            println!("Database process was terminated by a signal.");
        }

        println!("Restarting database process in 5 seconds...");
        thread::sleep(Duration::from_secs(5));
    }
}
```

Innhald frå Rust-filer

Fil: 3701d923/TTK4145_Real_Time_Programming-main/src/manager_process_pair.rs

```
// * NOTE: Some networking libraries requiring root privileges (for eks for pinging to router to read if we are still on the same network)
// * NOTE: Every new manager needs its own unique MANAGER_ID
// *
// * Because of these factors we must run this process as follows:
// * $ sudo -E MANAGER_ID=<ID> ELEVATOR_NETWORK_ID_LIST="[<ID 1>,<ID 2>,...,<ID N>]" cargo run --bin manager_process_pair

use std::env;
use std::process::{Command, ExitStatus};
use std::thread;
use std::time::Duration;

fn start_manager(manager_id: &str, elevator_network_id_list: &str) -> ExitStatus {
    println!(
        "Starting manager process with \n
        MANAGER_ID={} \n
        ELEVATOR_NETWORK_ID_LIST={} \n",
        manager_id, elevator_network_id_list
    );

    // Start the `cargo run` command with the necessary environment variable
    Command::new("sudo")
        .arg("-E")
        .env("MANAGER_ID", manager_id)
        .env("ELEVATOR_NETWORK_ID_LIST", elevator_network_id_list)
        .arg("cargo")
        .arg("run")
        .arg("--bin")
        .arg("manager") // Specify the database binary
        .status() // Run the command and return the ExitStatus
        .expect("Failed to start manager process") // Handle command failure
}

fn main() {
    // Ensure MANAGER_ID is passed to the parent process
    let manager_id = env::var("MANAGER_ID").expect("MANAGER_ID must be set");
    // Retrieve ELEVATOR_NETWORK_ID_LIST (defaulting to "[0]" if not set)
    let elevator_network_id_list = env::var("ELEVATOR_NETWORK_ID_LIST").unwrap_or_else(|_| "[0]".to_string());

    loop {
        // Start the child process and monitor its exit status
        let status = start_manager(&manager_id, &elevator_network_id_list);

        // Check if the process exited normally
        if let Some(code) = status.code() {
            println!("Manager process exited with code: {}", code);

            // Restart only if it didn't exit with a success code (0)
            if code == 0 {
                println!("Manager process exited successfully. Exiting monitor.");
                break;
            }
        } else {
            println!("Manager process was terminated by a signal.");
        }

        // Delay before restarting to avoid rapid restart loops
        println!("Restarting manager process in 5 seconds...");
        thread::sleep(Duration::from_secs(5));
    }
}
```

Innhald frå Rust-filer

Fil: 3701d923/TTK4145_Real_Time_Programming-main/src/elevator_logic/state_machine.rs

```
use std::collections::HashSet;

use super::utils::State;

pub fn handle_cab_request_current_floor(cab_queue: &HashSet<u8>, current_floor: u8) -> Option<State> {
    if cab_queue.contains(&current_floor) {
        return Some(State::Door); // Found a valid cab call
    } else {
        return None; // No cab call found
    }
}

pub fn handle_hall_up_request_current_floor(hall_up_queue: &HashSet<u8>, current_floor: u8) -> Option<State> {
    if hall_up_queue.contains(&current_floor) {
        return Some(State::Door); // Found a valid hall call
    } else {
        return None; // No hall call found
    }
}

pub fn handle_hall_down_request_current_floor(hall_down_queue: &HashSet<u8>, current_floor: u8) -> Option<State>
{
    if hall_down_queue.contains(&current_floor) {
        return Some(State::Door); // Found a valid hall call
    } else {
        return None; // No hall call found
    }
}

pub fn handle_cab_calls_while_moving(cab_queue: &HashSet<u8>, current_floor: u8, visited_floor: u8) ->
Option<State> {
    if cab_queue.contains(&current_floor) && visited_floor != current_floor {
        return Some(State::Door); // Found a valid cab call
    } else {
        return None; // No cab call found
    }
}

pub fn handle_up_requests(up_queue: &HashSet<u8>, current_floor: u8, visited_floor: u8) -> Option<State> {
    if up_queue.contains(&current_floor) && visited_floor != current_floor {
        return Some(State::Door); // Found a valid UP request
    } else {
        return None; // No UP request found
    }
}

pub fn handle_down_requests(down_queue: &HashSet<u8>, current_floor: u8, visited_floor: u8) -> Option<State> {
    if down_queue.contains(&current_floor) && visited_floor != current_floor {
        return Some(State::Door); // Found a valid DOWN request
    } else {
        return None; // No DOWN request found
    }
}

pub fn find_request_above(queue: &HashSet<u8>, current_floor: u8) -> Option<State> {
    if queue.iter().any(|&floor| floor > current_floor) {
        return Some(State::Up); // Found a request above
    } else {
        return None; // No requests above
    }
}

pub fn find_request_below(queue: &HashSet<u8>, current_floor: u8) -> Option<State> {
    if queue.iter().any(|&floor| floor < current_floor) {
        return Some(State::Down); // Found a request below
    } else {
        return None; // No requests below
    }
}

pub fn handle_emergency_stop(stop_button: bool) -> Option<State> {
    if stop_button {
        return Some(State::EmergencyStop);
    } else {
        return None;
    }
}

pub fn find_random_request(cab_queue: &HashSet<u8>, up_queue: &HashSet<u8>, down_queue: &HashSet<u8>, current_floor: u8) -> Option<State> {
    if let Some(&floor) = cab_queue.iter().next() {
        if floor > current_floor {
            return Some(State::Up);
        } else if floor < current_floor {
            return Some(State::Down);
        }
    }

    if let Some(&floor) = up_queue.iter().next() {
        if floor > current_floor {
            return Some(State::Up);
        } else if floor == current_floor {
            return Some(State::Door);
        } else if floor < current_floor {
            return Some(State::Down);
        }
    }

    if let Some(&floor) = down_queue.iter().next() {
        if floor > current_floor {
            return Some(State::Up);
        } else if floor == current_floor {
            return Some(State::Door);
        } else if floor < current_floor {
            return Some(State::Down);
        }
    }

    return None;
}
```

Innhald frå Rust-filer

Fil: 3701d923/TTK4145_Real_Time_Programming-main/src/elevator_logic/utils.rs

```
// Libraries for data structures
use serde::{Deserialize, Serialize};

#[derive(Copy, Clone, Debug, PartialEq, Eq, Serialize, Deserialize)]
pub enum State {
    Idle,
    Up,
    Down,
    Door,
    EmergencyStop,
    EmergencyStopIdle,
}

#[derive(Copy, Clone, Debug, PartialEq, Eq, Serialize, Deserialize)]
pub enum Direction {
    Stop,
    Up,
    Down,
}

#[derive(Serialize, Deserialize, Debug)]
pub struct ElevHallRequests {
    pub add_up: Option<u8>,
    pub add_down: Option<u8>,
    pub remove_up: Option<u8>,
    pub remove_down: Option<u8>,
}

// Function to create an `ElevHallRequests` and return its JSON representation
pub fn create_hall_request_json(add_up: Option<u8>, add_down: Option<u8>, remove_up: Option<u8>, remove_down:
Option<u8>) -> String {
    let request = ElevHallRequests { add_up, add_down, remove_up, remove_down };

    serde_json::to_string(&request).expect("Failed to serialize ElevHallRequests")
}
```

Innhald frå Rust-filer

Fil: 3701d923/TTK4145_Real_Time_Programming-main/src/elevator_io/driver.rs

```
use std::fmt;
use std::process;
use std::sync::Arc;
use tokio::io::{AsyncReadExt, AsyncWriteExt};
use tokio::net::TcpStream;
use tokio::sync::Mutex;

#[derive(Clone, Debug)]
pub struct Elevator {
    socket: Arc<Mutex<TcpStream>>,
    pub num_floors: u8,
}

pub const HALL_UP: u8 = 0;
pub const HALL_DOWN: u8 = 1;
pub const CAB: u8 = 2;

pub const DIRN_DOWN: u8 = u8::MAX;
pub const DIRN_STOP: u8 = 0;
pub const DIRN_UP: u8 = 1;

impl Elevator {
    /// Initialize the elevator by connecting to the given address
    pub async fn init(addr: &str, num_floors: u8) -> Elevator {
        match TcpStream::connect(addr).await {
            Ok(socket) => Self { socket: Arc::new(Mutex::new(socket)), num_floors },
            Err(err) => {
                eprintln!("Failed to connect to the elevator system: {}", err);
                process::exit(1);
            }
        }
    }

    /// Reload the elevator command
    pub async fn reload(&self) {
        let buf = [0, 0, 0, 0];
        if let Err(err) = self.send_command(&buf).await {
            self.log_error_and_exit("Failed to reload", err);
        }
    }

    /// Send motor direction command
    pub async fn motor_direction(&self, dirn: u8) {
        let buf = [1, dirn, 0, 0];
        if let Err(err) = self.send_command(&buf).await {
            self.log_error_and_exit("Failed to set motor direction", err);
        }
    }

    /// Set call button light
```

Innhald frå Rust-filer

```
pub async fn call_button_light(&self, floor: u8, call: u8, on: bool) {
    let buf = [2, call, floor, on as u8];
    if let Err(err) = self.send_command(&buf).await {
        self.log_error_and_exit("Failed to set call button light", err);
    }
}

/// Set floor indicator
pub async fn floor_indicator(&self, floor: u8) {
    let buf = [3, floor, 0, 0];
    if let Err(err) = self.send_command(&buf).await {
        self.log_error_and_exit("Failed to set floor indicator", err);
    }
}

/// Control the door light
pub async fn door_light(&self, on: bool) {
    let buf = [4, on as u8, 0, 0];
    if let Err(err) = self.send_command(&buf).await {
        self.log_error_and_exit("Failed to control door light", err);
    }
}

/// Control the stop button light
pub async fn stop_button_light(&self, on: bool) {
    let buf = [5, on as u8, 0, 0];
    if let Err(err) = self.send_command(&buf).await {
        self.log_error_and_exit("Failed to control stop button light", err);
    }
}

/// Check the status of a call button
pub async fn call_button(&self, floor: u8, call: u8) -> bool {
    let mut buf = [6, call, floor, 0];
    match self.send_and_receive(&mut buf).await {
        Ok(_) => buf[1] != 0,
        Err(err) => {
            self.log_error_and_exit("Failed to check call button", err);
            false // Unreachable, but required by compiler
        }
    }
}

/// Get the current floor sensor value
pub async fn floor_sensor(&self) -> Option<u8> {
    let mut buf = [7, 0, 0, 0];
    match self.send_and_receive(&mut buf).await {
        Ok(_) => {
            if buf[1] != 0 {
                Some(buf[2])
            } else {
                None
            }
        }
        Err(err) => {
            self.log_error_and_exit("Failed to read floor sensor", err);
            None // Unreachable, but required by compiler
        }
    }
}

/// Check the status of the stop button
pub async fn stop_button(&self) -> bool {
    let mut buf = [8, 0, 0, 0];
    match self.send_and_receive(&mut buf).await {
        Ok(_) => buf[1] != 0,
        Err(err) => {
            self.log_error_and_exit("Failed to check stop button", err);
            false // Unreachable, but required by compiler
        }
    }
}

/// Check the obstruction sensor
pub async fn obstruction(&self) -> bool {
    let mut buf = [9, 0, 0, 0];
    match self.send_and_receive(&mut buf).await {
        Ok(_) => buf[1] != 0,
        Err(err) => {
            self.log_error_and_exit("Failed to check obstruction sensor", err);
            false // Unreachable, but required by compiler
        }
    }
}

/// Helper method to send a command
async fn send_command(&self, buf: &[u8]) -> tokio::io::Result<()> {
    let mut sock = self.socket.lock().await;
    sock.write_all(buf).await // `write_all` already returns `Result<()>`
}

/// Helper method to send a command and receive a response
async fn send_and_receive(&self, buf: &mut [u8]) -> tokio::io::Result<()> {
    let mut sock = self.socket.lock().await;
    sock.write_all(buf).await?; // `write_all` ensures all bytes are sent
    sock.read_exact(buf).await.map(|_| ()) // Convert `Result<usize, _>` to `Result<(), _>`
}

/// Helper method to log an error and terminate the process
fn log_error_and_exit(&self, msg: &str, err: tokio::io::Error) {
    eprintln!();
    eprintln!("#============================================================#");
    eprintln!("ERROR: {}: {}", msg, err);
    eprintln!("#============================================================#");
```

```
        eprintln!();
        process::exit(1);
    }
}

impl fmt::Display for Elevator {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        let addr = tokio::task::block_in_place(|| {
            let rt = tokio::runtime::Handle::current();
            rt.block_on(async {
                let sock = self.socket.lock().await;
                sock.peer_addr()
            })
        });

        match addr {
            Ok(addr) => write!(f, "Elevator@{}({})", addr, self.num_floors),
            Err(_) => write!(f, "Elevator@(unknown)({})", self.num_floors),
        }
    }
}
```

Innhald frå Rust-filer

Fil: 3701d923/TTK4145_Real_Time_Programming-main/src/elevator_io/data.rs

```rust
use super::driver;
use tokio::sync::mpsc;
use tokio::time::{self, Duration};

#[derive(Debug)]
pub struct CallButton {
    pub floor: u8,
    pub call: u8,
}

pub async fn call_buttons(elev: driver::Elevator, ch: mpsc::Sender<CallButton>, period: Duration) {
    let mut prev = vec![[false; 3]; elev.num_floors.into()];
    let mut interval = time::interval(period);

    loop {
        interval.tick().await; // Ensure periodic execution
        for f in 0..elev.num_floors {
            for c in 0..3 {
                let v = elev.call_button(f, c).await; // Directly returns a bool
                if v && prev[f as usize][c as usize] != v {
                    if ch.send(CallButton { floor: f, call: c }).await.is_err() {
                        eprintln!("Failed to send CallButton update");
                        return;
                    }
                }
                prev[f as usize][c as usize] = v;
            }
        }
    }
}

pub async fn floor_sensor(elev: driver::Elevator, ch: mpsc::Sender<u8>, period: Duration) {
    let mut interval = time::interval(period);

    loop {
        interval.tick().await;
        if let Some(f) = elev.floor_sensor().await {
            if ch.send(f).await.is_err() {
                eprintln!("Failed to send floor sensor update");
                let _prev = 0;
            }
            let _prev = f;
        }
    }
}

pub async fn stop_button(elev: driver::Elevator, ch: mpsc::Sender<bool>, period: Duration) {
    let mut prev = false; // Previous stop button state
    let mut interval = time::interval(period);
    loop {
        interval.tick().await;

        // Directly fetch the stop button state (returns a bool)
        let v = elev.stop_button().await;
        if v != prev {
            if ch.send(v).await.is_err() {
                eprintln!("Failed to send stop button update");
                return;
            }
            prev = v;
        }
    }
}

pub async fn obstruction(elev: driver::Elevator, ch: mpsc::Sender<bool>, period: Duration) {
    let mut prev = false; // Previous obstruction state
    let mut interval = time::interval(period);

    loop {
        interval.tick().await;

        // Directly fetch the obstruction state (returns a bool)
        let v = elev.obstruction().await;
        if v != prev {
            if ch.send(v).await.is_err() {
                eprintln!("Failed to send obstruction update");
                return;
            }
            prev = v;
        }
    }
}
```

Fil: 3701d923/TTK4145_Real_Time_Programming-main/src/elevator_io/driver_sync.rs

```
// This is a special case of the normal driver
// This is made for synchronous processes
// For example peer process

use std::fmt;
use std::io::Write;
use std::net::TcpStream;
use std::process;
use std::sync::{Arc, Mutex};

#[derive(Clone, Debug)]
pub struct Elevator {
    socket: Arc<Mutex<TcpStream>>,
    pub num_floors: u8,
}

pub const DIR_DOWN: u8 = u8::MAX;
pub const DIR_STOP: u8 = 0;
pub const DIR_UP: u8 = 1;

impl Elevator {
    /// Initialize the elevator by connecting to the given address
    pub fn init_sync(addr: &str, num_floors: u8) -> Elevator {
        match TcpStream::connect(addr) {
            Ok(socket) => Self { socket: Arc::new(Mutex::new(socket)), num_floors },
            Err(err) => {
                eprintln!("Failed to connect to the elevator system: {}", err);
                process::exit(1);
            }
        }
    }

    /// Send motor direction command (synchronous version)
    pub fn motor_direction_sync(&self, dirn: u8) {
        let buf = [1, dirn, 0, 0];
        if let Err(err) = self.send_command_sync(&buf) {
            self.log_error_and_exit("Failed to set motor direction", err);
        }
    }

    /// Synchronous helper method to send a command
    fn send_command_sync(&self, buf: &[u8]) -> std::io::Result<()> {
        let mut sock = self.socket.lock().unwrap();
        sock.write_all(buf)?; // Write all bytes synchronously
        Ok(())
    }

    /// Helper method to log an error and terminate the process
    fn log_error_and_exit(&self, msg: &str, err: std::io::Error) {
        eprintln!();
        eprintln!("#============================================================#");
        eprintln!("ERROR: {}: {}", msg, err);
        eprintln!("#============================================================#");
        eprintln!();
        process::exit(1);
    }
}

impl fmt::Display for Elevator {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        let addr = self.socket.lock().ok().and_then(|sock| sock.peer_addr().ok());
        match addr {
            Some(addr) => write!(f, "Elevator@{}({})", addr, self.num_floors),
            None => write!(f, "Elevator@(unknown)({})", self.num_floors),
        }
    }
}
```

Innhald frå Rust-filer

Fil: 3701d923/TTK4145_Real_Time_Programming-main/src/distributed_systems/utils.rs

```
// Libraries for network status and diagnostics
// * NOTE: The `tokio_icmp_echo` library requires ICMP sockets, which are ONLY available for root users.
// *       Therefore, we must run this program as `sudo cargo run`.
// *       The library is used to ping the router for network connectivity status.
use futures::{future, StreamExt};
use std::collections::HashSet;
use std::net::IpAddr;
use std::process::Command;
use tokio_icmp_echo::Pinger;

// Libraries for Distributed Networks
use zenoh::sample::Sample;

// Libraries for asynchronous multithreaded activities
use tokio::time::{timeout, Duration};

// Library for formatting
use crate::elevator_logic::utils::{Direction, State};
use serde::{Deserialize, Serialize};
use serde_json::from_str;
use std::collections::HashMap;

//
=====================================================================================
=========
// Parses the payload of a Zenoh `Sample` message into a `String`.
//
// - Converts the payload of a Zenoh `Sample` to a UTF-8 `String`.
// - If the payload is invalid UTF-8, it defaults to "Invalid UTF-8".
// - Ensures the final output is always a `String`.
//
// Arguments:
// - `message`: The Zenoh `Sample` containing the payload to parse.
//
// Returns:
// - A `String` representation of the payload.
//
// Example:
// let parsed_message = parse_message(sample);
// println!("{}", parsed_message); // Outputs the payload as a `String`.
//
=====================================================================================
=========
pub fn parse_message_to_string(message: Sample) -> String {
    return message.payload().try_to_string().unwrap_or_else(|_| "Invalid UTF-8".into()).to_string();
    // Convert Cow<'_, str> to String
}

/// Parses a Zenoh message payload into a `HashSet<u8>`.
///
```

```
/// - Removes curly braces `{}` from the payload.
/// - Assumes the payload is a comma-separated list of integers (e.g., "1,2,3").
/// - Skips invalid entries.
///
/// Returns:
/// - `HashSet<u8>` containing the parsed integers.
pub fn parse_message_to_hashset_u8(message: Sample) -> HashSet<u8> {
    // Convert the payload to a string
    let payload = message.payload().try_to_string().unwrap_or_else(|_| "".into()); // Default to empty string on error

    // Remove the outer curly braces (if any)
    let payload = payload.trim().trim_start_matches('{').trim_end_matches('}');

    // Parse the cleaned payload into a HashSet<u8>
    let parsed_set: HashSet<u8> = payload
        .split(',') // Split the string by commas
        .filter_map(|s| {
            let trimmed = s.trim();
            match trimmed.parse::<u8>() {
                Ok(value) => Some(value), // Valid integer
                Err(_) => {
                    println!("Skipping invalid entry: {}", trimmed); // Debug invalid entries
                    None // Skip invalid entries
                }
            }
        })
        .collect(); // Collect valid values into a HashSet

    parsed_set
}

// A data structure to backup data and parse it with the helper function
#[derive(Debug, Serialize, Deserialize)]
pub struct ElevatorBackup {
    pub state: State,
    pub direction: Direction,
    pub current_floor: Option<u8>,
    pub cab_queue: HashSet<u8>,
    pub hall_up_queue: HashSet<u8>,
    pub hall_down_queue: HashSet<u8>,
}

pub fn parse_message_to_elevator_backup(message: Sample) -> Option<ElevatorBackup> {
    message
        .payload()
        .try_to_string()
        .ok()
        .and_then(|json_str| from_str::<ElevatorBackup>(&json_str).ok())
}

// A way to convert message we receive from manager node to a understandable format
#[derive(Debug, Serialize, Deserialize)]
```

Innhald frå Rust-filer

```
pub struct ElevatorRequests {
    pub requests: HashMap<String, Vec<Vec<bool>>>, // Elevator ID -> 2D bool array
}

pub fn parse_message_to_elevator_requests(message: Sample) -> Option<ElevatorRequests> {
    message
        .payload()
        .try_to_string()
        .ok()
        .and_then(|json_str| from_str::<HashMap<String, Vec<Vec<bool>>>>(&json_str).ok())
        .map(|parsed| ElevatorRequests { requests: parsed })
}

//
==========================================================================================
=========
// Awaits a future with a timeout.
//
// - Waits for a future to complete within a specified timeout duration.
// - If the future completes successfully within the timeout, returns `Some` with the result.
// - If the timeout is exceeded or the future errors out, returns `None`.
//
// Arguments:
// - `duration_ms`: The timeout duration in milliseconds.
// - `future`: The future to await, which must return `Result<T, zenoh::Error>`.
//
// Returns:
// - `Option<T>`: `Some` if the future completes successfully within the timeout, `None` otherwise.
//
// Example:
// let result = wait_with_timeout(5000, some_async_operation()).await;
// if let Some(value) = result {
//     println!("Operation succeeded: {:?}", value);
// } else {
//     println!("Operation timed out.");
// }
//
==========================================================================================
=========
pub async fn wait_with_timeout<T>(duration_ms: u64, future: impl futures::Future<Output = Result<T, zenoh::Error>>)
-> Option<T> {
    let duration = Duration::from_millis(duration_ms);
    return timeout(duration, future).await.ok().and_then(|res| res.ok());
}

//
==========================================================================================
=========
// Retrieves the router's IP address.
//
// - Executes the `ip route` command to find the default gateway (router).
// - Extracts and parses the IP address from the command's output.
// - If the command fails or no default gateway is found, returns `None`.
//
// Returns:
// - `Option<IpAddr>`: The router's IP address if found, or `None` otherwise.
//
// Example:
// if let Some(router_ip) = get_router_ip().await {
//     println!("Router IP: {}", router_ip);
// } else {
//     println!("Failed to find the router IP.");
// }
//
============================================================================================
=========
pub async fn get_router_ip() -> Option<IpAddr> {
    // Run the `ip route` command and capture the output
    let output = Command::new("ip").arg("route").output().expect("Failed to execute ip route command");

    // Check if the command executed successfully
    if !output.status.success() {
        println!("get_router_ip(): Command Failed!");
        return None;
    }

    // Parse the command output to find the default gateway
    let stdout = String::from_utf8_lossy(&output.stdout);
    for line in stdout.lines() {
        if line.starts_with("default via") {
            if let Some(ip_str) = line.split_whitespace().nth(2) {
                return ip_str.parse().ok(); // Parse the IP address
            }
        }
    }

    println!("get_router_ip(): No default gateway found!");
    return None;
}

//
============================================================================================
=========
// Enum representing the network status of the node.
//
// Variants:
// - `Connected`: Indicates the node is connected to the network (at least one ping succeeded).
// - `Disconnected`: Indicates the node is disconnected (all pings failed or an error occurred).
//
// Debug trait is derived to allow easy debugging with `println!("{:?}", NetworkStatus::Connected)`.
//
============================================================================================
=========
#[derive(Debug)]
```

Innhald frå Rust-filer

```
pub enum NetworkStatus {
    Connected,
    Disconnected,
}

//
============================================================================================
=========
// Checks the network connectivity to the router.
//
// - Sends ICMP echo requests (pings) to the router's IP address.
// - Tracks whether at least one ping succeeds to determine connectivity status.
// - Uses the `tokio_icmp_echo` library for non-blocking pings.
//
// Arguments:
// - `router_ip`: The IP address of the router to ping.
//
// Returns:
// - `NetworkStatus`: `Connected` if at least one ping succeeds, `Disconnected` otherwise.
//
// Example:
// let status = network_status(router_ip).await;
// match status {
//     NetworkStatus::Connected => println!("Network is connected."),
//     NetworkStatus::Disconnected => println!("Network is disconnected."),
// }
//
============================================================================================
=========
pub async fn network_status(router_ip: IpAddr) -> NetworkStatus {
    // Create a new pinger instance
    let pinger = match Pinger::new().await {
        Ok(p) => p,
        Err(_) => return NetworkStatus::Disconnected, // Assume disconnected if pinger setup fails
    };

    // Create a stream for sending ICMP packets to the router IP
    let stream = pinger.chain(router_ip).stream();

    // Set the number of ICMP echo requests to send (number of tries)
    let tries = 5;

    // Track whether at least one ping succeeds
    let mut is_connected = false;

    // Process up to `tries` number of ping responses
    stream
        .take(tries)
        .for_each(|mb_time| {
            match mb_time {
                Ok(Some(_)) => is_connected = true, // Mark as connected on successful ping
                Ok(None) => {}                      // Do nothing on timeout
                Err(_) => {}                              // Do nothing on error
            }
            future::ready(())
        })
        .await;

    // Return the appropriate network status
    if is_connected {
        return NetworkStatus::Connected;
    } else {
        return NetworkStatus::Disconnected;
    }
}
```

Innhald frå Rust-filer

Fil: 3701d923/TTK4145_Real_Time_Programming-main/src/elevator_algorithm/cost_algorithm.rs

```
// Library that allows us to use environment variables or command-line arguments to pass variables from terminal to the program directly
use std::env;

// Library for executing terminal commands
use tokio::process::Command;

// Function to execute the algorithm
pub async fn run_cost_algorithm(json_str: String) -> String {
    let algorithm_path = env::current_dir()
        .unwrap()
        .join("Project-resources")
        .join("cost_fns")
        .join("hall_request_assigner")
        .join("hall_request_assigner");

    let output = Command::new(algorithm_path)
        .arg("--input")
        .arg(json_str) //  Use JSON directly as received
        .output()
        .await
        .expect("Failed to start algorithm");

    let output_str = String::from_utf8_lossy(&output.stdout).into_owned();

    return output_str;
}
```

Innhald frå Rust-filer

Fil: 3701d923/TTK4145_Real_Time_Programming-main/src/elevator_algorithm/utils.rs

```
use serde::{Deserialize, Serialize};
use std::collections::HashMap;

#[allow(non_snake_case)]
#[derive(Serialize, Deserialize, Clone)]
pub struct ElevState {
    pub behaviour: String,
    pub floor: u8,
    pub direction: String,
    pub cabRequests: Vec<bool>,
}

impl ElevState {
    pub fn from_elevmsg(msg: ElevMsg) -> (Vec<Vec<bool>>, ElevState) {
        (
            msg.hallRequests,
             ElevState { behaviour: msg.behaviour, floor: msg.floor, direction: msg.direction, cabRequests: msg.cabRequests
},
        )
    }
}

#[allow(non_snake_case)]
#[derive(Serialize, Deserialize, Clone)]
pub struct AlgoInput {
    pub hallRequests: Vec<Vec<bool>>,
    pub states: HashMap<String, ElevState>,
}

impl AlgoInput {
    pub fn new() -> Self {
        AlgoInput { hallRequests: vec![], states: HashMap::new() }
    }
}

#[allow(non_snake_case)]
#[derive(Serialize, Deserialize)]
pub struct ElevMsg {
    pub hallRequests: Vec<Vec<bool>>,
    pub behaviour: String,
    pub floor: u8,
    pub direction: String,
    pub cabRequests: Vec<bool>,
}
```