Innhald frå Rust-filer

Fil: elevator_pro_rebrand/src/init.rs

```rust
use std::{borrow::Cow, env, net::SocketAddr, sync::atomic::Ordering, time::Duration};
use tokio::{net::UdpSocket, time::{sleep, timeout, Instant}};
use socket2::{Domain, Socket, Type};
use local_ip_address::local_ip;
use crate::{config, ip_help_functions::ip2id, network, print, world_view::{self, serial, ElevatorContainer, WorldView}};
use tokio::process::Command;

/// ### Initializes the worldview on startup
///
/// This function creates an initial worldview for the elevator system and attempts to join an existing network if possible.
///
/// ## Steps:
/// 1. **Create an empty worldview and elevator container.**
/// 2. **Add an initial placeholder task** to both the task queue and task status list.
/// 3. **Retrieve the local machine's IP address** to determine its unique ID.
/// 4. **Set the elevator ID and master ID** using the extracted IP-based identifier.
/// 5. **Listen for UDP messages** for a brief period to detect other nodes on the network.
/// 6. **If no nodes are found**, return the current worldview as is, with self id as the network master.
/// 7. **If other elevators are detected**, merge their worldview with the local elevator's data.
/// 8. **Check if the master ID should be updated** based on the smallest ID present.
/// 9. **Return the serialized worldview**, ready to be used for network synchronization.
///
/// ## Returns:
/// - A `Vec<u8>` containing the serialized worldview data.
///
/// ## Panics:
/// - No internet connection on start-up will result in a panic!
///
/// ## Example Usage:
/// ```rust
/// let worldview_data: Vec<u8> = initialize_worldview().await;
/// let worldview: worldview::WorldView = worldview::serial::deserialize_worldview(&worldview_data);
/// ```
pub async fn initialize_worldview(self_container : Option< world_view::ElevatorContainer>) -> Vec<u8> {
    let mut worldview = WorldView::default();

    let mut elev_container = if let Some(container) = self_container {
        container
    } else {
        // Opprett ein standard ElevatorContainer med ein initial placeholder-task
        let container = ElevatorContainer::default();
        container
    };



    // Retrieve local IP address
    let ip = match local_ip() {
        Ok(ip) => ip,
        Err(e) => {
            print::err(format!("Failed to get local IP at startup: {}", e));
```

```
            panic!();
        }
    };

    // Extract self ID from IP address (last segment of IP)
    network::set_self_id(ip2id(ip));
    elev_container.elevator_id = network::read_self_id();
    worldview.master_id = network::read_self_id();
    worldview.add_elev(elev_container.clone());

    // Listen for UDP messages for a short time to detect other elevators
    let wv_from_udp = check_for_udp().await;
    if wv_from_udp.is_empty() {
        print::info("No other elevators detected on the network.".to_string());
        return serial::serialize_worldview(&worldview);
    }

    // If other elevators are found, merge worldview and add the local elevator
    let mut wv_from_udp_deser = serial::deserialize_worldview(&wv_from_udp);

    // Check if the network has backed up any cab_requests from you, save them if that is the case
    let saved_cab_requests: std::collections::HashMap<u8, Vec<bool>> = wv_from_udp_deser.cab_requests_backup.clone();
    if let Some(saved_requests) = saved_cab_requests.get(&elev_container.elevator_id) {
        elev_container.cab_requests = saved_requests.clone();
    }
    // Add your elevator to the worldview
    wv_from_udp_deser.add_elev(elev_container.clone());

    // Set self as master if the current master has a higher ID
    if wv_from_udp_deser.master_id > network::read_self_id() {
        wv_from_udp_deser.master_id = network::read_self_id();
    }

    // Serialize and return the updated worldview
    serial::serialize_worldview(&wv_from_udp_deser)
}



/// ### Listens for a UDP broadcast message for 1 second
///
/// This function listens for incoming UDP broadcasts on a predefined port.
/// It ensures that the received message originates from the expected network before accepting it.
///
/// ## Steps:
/// 1. **Set up a UDP socket** bound to a predefined broadcast address.
/// 2. **Configure socket options** for reuse and broadcasting.
/// 3. **Start a timer** and listen for UDP packets for up to 1 second.
/// 4. **If a message is received**, attempt to decode it as a UTF-8 string.
/// 5. **Filter out messages that do not contain the expected key**.
/// 6. **Extract the relevant data** and convert it into a `Vec<u8>`.
```

```rust
/// 7. **Return the parsed data or an empty vector** if no valid message was received.
///
/// ## Returns:
/// - A `Vec<u8>` containing parsed worldview data if a valid UDP message was received.
/// - An empty vector if no message was received within 1 second.
///
/// ## Example Usage:
/// ```rust
/// let udp_data = check_for_udp().await;
/// if !udp_data.is_empty() {
///     println!("Received worldview data: {:?}", udp_data);
/// } else {
///     println!("No UDP message received within 1 second.");
/// }
/// ```
pub async fn check_for_udp() -> Vec<u8> {
    // Construct the UDP broadcast listening address
    let broadcast_listen_addr = format!("{}:{}", config::BC_LISTEN_ADDR, config::DUMMY_PORT);
    let socket_addr: SocketAddr = broadcast_listen_addr.parse().expect("Invalid address");

    // Create a new UDP socket
    let socket_temp = Socket::new(Domain::IPV4, Type::DGRAM, None)
        .expect("Failed to create new socket");

    // Configure socket for address reuse and broadcasting
    socket_temp.set_nonblocking(true).expect("Failed to set non-blocking");
    socket_temp.set_reuse_address(true).expect("Failed to set reuse address");
    socket_temp.set_broadcast(true).expect("Failed to enable broadcast mode");
    socket_temp.bind(&socket_addr.into()).expect("Failed to bind socket");

    // Convert standard socket into an async UDP socket
    let socket = UdpSocket::from_std(socket_temp.into()).expect("Failed to create UDP socket");

    // Buffer for receiving UDP data
    let mut buf = [0; config::UDP_BUFFER];
    let mut read_wv: Vec<u8> = Vec::new();

    // Placeholder for received message
    let mut message: Cow<'_, str>;

    // Start the timer for 1-second listening duration
    let time_start = Instant::now();
    let duration = Duration::from_secs(1);

    while Instant::now().duration_since(time_start) < duration {
        // Attempt to receive a UDP packet within the timeout duration
        let recv_result = timeout(duration, socket.recv_from(&mut buf)).await;

        match recv_result {
            Ok(Ok((len, _))) => {
                // Convert the received bytes into a string
                message = String::from_utf8_lossy(&buf[..len]).into_owned().into();
```

```
            }
            Ok(Err(e)) => {
                // Log errors if receiving fails
                print::err(format!("init.rs, udp_listener(): {}", e));
                continue;
            }
            Err(_) => {
                // Timeout occurred  no data received within 1 second
                print::warn("Timeout - no data received within 1 second.".to_string());
                break;
            }
        }

        // Verify that the UDP message is from our expected network
        if &message[1..config::KEY_STR.len() + 1] == config::KEY_STR {
            // Extract and clean the message by removing the key and surrounding characters
            let clean_message = &message[config::KEY_STR.len() + 3..message.len() - 1];

            // Parse the message as a comma-separated list of u8 values
            read_wv = clean_message
                .split(", ") // Split on ", "
                .filter_map(|s| s.parse::<u8>().ok()) // Convert to u8, ignore errors
                .collect(); // Collect into a Vec<u8>

            break; // Exit loop as a valid message was received
        }
    }

    // Drop the socket to free resources
    drop(socket);

    // Return the parsed UDP message data
    read_wv
}


/// ### Reads arguments from `cargo run`
///
/// Used to modify what is printed during runtime. Available options:
///
/// `print_wv::(true/false)` &rarr; Prints the worldview twice per second
/// `print_err::(true/false)` &rarr; Prints error messages
/// `print_wrn::(true/false)` &rarr; Prints warning messages
/// `print_ok::(true/false)` &rarr; Prints OK messages
/// `print_info::(true/false)` &rarr; Prints informational messages
/// `print_else::(true/false)` &rarr; Prints other messages, including master, slave, and color messages
/// `debug::` &rarr; Disables all prints except error messages
/// `help` &rarr; Displays all possible arguments without starting the program
///
/// If no arguments are provided, all prints are enabled by default.
///
/// Secret options:
```

Innhald frå Rust-filer

```
/// `backup` &rarr; Starts the program in backup-mode.
///
pub fn parse_args() -> bool {
    let args: Vec<String> = env::args().collect();

    // Hvis det ikke finnes argumenter, returner false
    if args.len() <= 0 {
        return false;
    }

    for arg in &args[1..] {
        let parts: Vec<&str> = arg.split("::").collect();
        if parts.len() == 2 {
            let key = parts[0].to_lowercase();
            let value = parts[1].to_lowercase();
            let is_true = value == "true";


            match key.as_str() {
                "print_wv" => *config::PRINT_WV_ON.lock().unwrap() = is_true,
                "print_err" => *config::PRINT_ERR_ON.lock().unwrap() = is_true,
                "print_warn" => *config::PRINT_WARN_ON.lock().unwrap() = is_true,
                "print_ok" => *config::PRINT_OK_ON.lock().unwrap() = is_true,
                "print_info" => *config::PRINT_INFO_ON.lock().unwrap() = is_true,
                "print_else" => *config::PRINT_ELSE_ON.lock().unwrap() = is_true,
                "debug" => { // Debug modus: Kun error-meldingar
                    *config::PRINT_WV_ON.lock().unwrap() = false;
                    *config::PRINT_WARN_ON.lock().unwrap() = false;
                    *config::PRINT_OK_ON.lock().unwrap() = false;
                    *config::PRINT_INFO_ON.lock().unwrap() = false;
                    *config::PRINT_ELSE_ON.lock().unwrap() = false;
                }
                _ => {}
            }

        } else if arg.to_lowercase() == "help" {
            println!("Tilgjengelige argument:");
            println!("  print_wv::true/false");
            println!("  print_err::true/false");
            println!("  print_warn::true/false");
            println!("  print_ok::true/false");
            println!("  print_info::true/false");
            println!("  print_else::true/false");
            println!("  debug (kun error-meldingar vises)");
            println!("  backup (starter backup-prosess)");
            std::process::exit(0);
        } else if arg.to_lowercase() == "backup" {
            return true;
        }
    }

    // Hvis ingen av argumentene matcher "backup", returner false
    false
}


/// Returns the terminal command for the corresponding OS.
///
/// # Example
/// ```
/// use elevatorpro::utils::get_terminal_command;
///
/// let (cmd, args) = get_terminal_command();
///
/// if cfg!(target_os = "windows") {
///     assert_eq!(cmd, "cmd");
///     assert_eq!(args, vec!["/C", "start"]);
/// } else {
///     assert_eq!(cmd, "gnome-terminal");
///     assert_eq!(args, vec!["--"]);
/// }
/// ```
pub fn get_terminal_command() -> (String, Vec<String>) {
    if cfg!(target_os = "windows") {
        ("cmd".to_string(), vec!["/C".to_string(), "start".to_string()])
    } else {
        ("gnome-terminal".to_string(), vec!["--".to_string()])
    }
}



/// ### Executes the `build.sh` script for the hall_request_assigner cost function in a separate process.
///
/// This function asynchronously runs the `build.sh` script located in the
/// `libs/Project_resources/cost_fns/hall_request_assigner` directory using `bash`.
///
/// If the build script runs successfully, its stdout and stderr are printed to the console,
/// and the program continues normally. If the script fails, the function will print relevant
/// error output, suggest manual build steps for debugging, and then terminate the process
/// by panicking.
///
/// # Panics
/// Panics if the script fails to execute or if it exits with a non-zero status code.
/// This ensures the caller is alerted early to any build issues.
pub async fn build_cost_fn() {

    let output = Command::new("bash")
        .arg("build.sh")
        .current_dir("libs/Project_resources/cost_fns/hall_request_assigner")
        .output()
        .await
        .expect("Klarte ikkje starte build.sh");
```

```
    println!("stdout: {}", String::from_utf8_lossy(&output.stdout));
    eprintln!("stderr: {}", String::from_utf8_lossy(&output.stderr));

    if output.status.success() {
        println!("build.sh completed successfully.");
    } else {
        eprintln!("build.sh failed. Please try building manually in a new terminal:");
        eprintln!("1. cd libs/Project_resources/cost_fns/hall_request_assigner");
        eprintln!("2. bash build.sh");
        panic!("Failed to build hall_request_assigner.");
    }
    sleep(Duration::from_millis(2000)).await;
}
```

Innhald frå Rust-filer

Fil: elevator_pro_rebrand/src/main.rs

```
use tokio::sync::mpsc;
use tokio::net::TcpStream;
use std::net::SocketAddr;

use elevatorpro::{backup, elevator_logic, manager, network::{self, local_network, tcp_network, udp_network},
world_view};
use elevatorpro::init;
use elevatorpro::print;






#[tokio::main]
async fn main() {
    // Sjekk om programmet startes som backup, retunerer true visst den blei det
    // vi starter i bacup med å skrive "cargo r -- backup"
    let is_backup = init::parse_args();

    let mut self_container: Option< world_view::ElevatorContainer> = None;
    if is_backup {
        println!("Starter backup-prosess...");
        self_container = backup::run_as_backup().await;
    }

    // Hvis vi ikke er backup, starter vi som master! eller om vi kjem ut, så tar vi over


    init::build_cost_fn().await;
    // Vanlig hovedprosess starter her:
    print::info("Starter hovedprosess...".to_string());





    /*Skaper oss eit verdensbildet ved fødselen, vi tar vår første pust */
    let worldview_serialised = init::initialize_worldview(self_container).await;



    /* START ----------- Init av channels brukt til oppdatering av lokal worldview ---------------------- */
    let main_mpscs = local_network::Mpscs::new();
    let watches = local_network::Watches::new();

    // Send the initialized worldview on the worldview watch, so its not empty when rx tries to borrow it
    let _ = watches.txs.wv.send(worldview_serialised.clone());
    // Seperate the watch Tx's so they can be sent to theis designated tasks
    let wv_watch_tx = watches.txs.wv;
    // let elev_task_tx= watches.txs.elev_task;
    // Seperate the mpsc Rx's so they can be sent to [local_network::update_wv_watch]
    let mpsc_rxs = main_mpscs.rxs;
    // Seperate the mpsc Tx's so they can be sent to their designated tasks
    let elevator_states_tx = main_mpscs.txs.elevator_states;
    let delegated_tasks_tx = main_mpscs.txs.delegated_tasks;
    let udp_wv_tx = main_mpscs.txs.udp_wv;
    let remove_container_tx = main_mpscs.txs.remove_container;
    let container_tx = main_mpscs.txs.container;
    let connection_to_master_failed_tx_clone = main_mpscs.txs.connection_to_master_failed.clone();
    let sent_tcp_container_tx = main_mpscs.txs.sent_tcp_container;
    let connection_to_master_failed_tx = main_mpscs.txs.connection_to_master_failed;
    let new_wv_after_offline_tx = main_mpscs.txs.new_wv_after_offline;

    /* SLUTT ----------- Init av channels brukt til oppdatering av lokal worldview ---------------------- */

    /* START ----------- Task for å overvake Nettverksstatus ---------------------- */
    {
        let wv_watch_rx = watches.rxs.wv.clone();
        let _network_status_watcher_task = tokio::spawn(async move {
            print::info("Starter å passe på nettverket".to_string());
            let _ = network::watch_ethernet(wv_watch_rx, new_wv_after_offline_tx).await;
        });
    }
    /* SLUTT ----------- Task for å overvake Nettverksstatus ---------------------- */


    /* START ----------- Init av diverse channels ---------------------- */
    // Create other channels used for other things
    let (socket_tx, socket_rx) = mpsc::channel::<(TcpStream, SocketAddr)>(100);

/* SLUTT ----------- Init av diverse channels ---------------------- */

/* START ----------- Starte kritiske tasks ----------- */
    {
        //Task som kontinuerlig oppdaterer lokale worldview
        let _update_wv_task = tokio::spawn(async move {
            print::info("Starter å oppdatere wv".to_string());
            let _ = local_network::update_wv_watch(mpsc_rxs, wv_watch_tx, worldview_serialised).await;
        });
    }
    //Task som håndterer den lokale heisen
    //TODO: Få den til å signalisere at vi er i known state.
    {
        let wv_watch_rx = watches.rxs.wv.clone();
        let _local_elev_task = tokio::spawn(async move {
            let _ = elevator_logic::run_local_elevator(wv_watch_rx, elevator_states_tx).await;
        });
    }
    {
        let wv_watch_rx = watches.rxs.wv.clone();
        let _manager_task = tokio::spawn(async move {
            print::info("Staring task manager".to_string());
            let _ = manager::start_manager(wv_watch_rx, delegated_tasks_tx).await;
        });
    }
/* SLUTT ----------- Starte kritiske tasks ----------- */

    // Start backup server i en egen task
    {
        let wv_watch_rx = watches.rxs.wv.clone();
        let _backup_task = tokio::spawn(async move {
            print::info("Starter backup".to_string());
            tokio::spawn(backup::start_backup_server(wv_watch_rx));
        });
    }

/* START ----------- Starte Eksterne Nettverkstasks --------------------- */
    //Task som hører etter UDP-broadcasts
    {
        let wv_watch_rx = watches.rxs.wv.clone();
        let _listen_task = tokio::spawn(async move {
            print::info("Starter å høre etter UDP-broadcast".to_string());
            let _ = udp_network::start_udp_listener(wv_watch_rx, udp_wv_tx).await;
        });
    }

    //Task som starter egen UDP-broadcaster
    {
        let wv_watch_rx = watches.rxs.wv.clone();
        let _broadcast_task = tokio::spawn(async move {
            print::info("Starter UDP-broadcaster".to_string());
            let _ = udp_network::start_udp_broadcaster(wv_watch_rx).await;
        });
    }

    //Task som håndterer TCP-koblinger
    {
        let wv_watch_rx = watches.rxs.wv.clone();
        let _tcp_task = tokio::spawn(async move {
            print::info("Starter å TCPe".to_string());
                                    let _ = tcp_network::tcp_handler(wv_watch_rx, remove_container_tx, container_tx,
connection_to_master_failed_tx, sent_tcp_container_tx, socket_rx).await;
        });
    }

    //UDP Watchdog
    {
        let _udp_watchdog = tokio::spawn(async move {
            print::info("Starter udp watchdog".to_string());
            let _ = udp_network::udp_watchdog(connection_to_master_failed_tx_clone).await;
        });
    }

    //Task som starter TCP-listener
    {
        let _listener_handle = tokio::spawn(async move {
            print::info("Starter tcp listener".to_string());
            let _ = tcp_network::listener_task(socket_tx).await;
        });
    }
    // Lag prat med egen heis thread her
/* SLUTT ----------- Starte Eksterne Nettverkstasks ---------------------- */


    // Task som printer worldview
    // let _print_task = tokio::spawn(async move {
    //     let mut wv = world_view::get_wv(watches.rxs.wv.clone());
    //     loop {
    //         if world_view::update_wv(watches.rxs.wv.clone(), &mut wv).await {
    //             print::worldview(wv.clone());
    //             tokio::time::sleep(Duration::from_millis(500)).await;
    //         }
    //     }
    // });

    //Vent med å avslutte programmet
    loop{
        tokio::task::yield_now().await;
    }
}
```

Innhald frå Rust-filer

Fil: elevator_pro_rebrand/src/ip_help_functions.rs

```rust
use std::net::IpAddr;
use std::u8;
use crate::config;



/// Extracts your ID based on `ip`
///
/// ## Example
/// ```
/// let id = id_fra_ip("a.b.c.d:e");
/// ```
/// returnerer d
///
pub fn ip2id(ip: IpAddr) -> u8 {
    let ip_str = ip.to_string();
    let mut ip_int = config::ERROR_ID;
    let id_str = ip_str.split('.')           // Del på punktum
        .nth(3)              // Hent den 4. delen (d)
        .and_then(|s| s.split(':')  // Del på kolon hvis det er en port etter IP-en
            .next())         // Ta kun første delen før kolon
        .and_then(|s| s.parse::<u8>().ok());  // Forsøk å parse til u8

    match id_str {
        Some(value) => {
            ip_int = value;
        }
        None => {
            println!("Ingen gyldig ID funnet. (konsulent.rs, id_fra_ip())");
        }
    }
    ip_int
}

/// Extracts the root part of an IP address (removes the last segment).
///
/// ## Example
/// ```
/// use std::net::IpAddr;
/// use std::str::FromStr;
/// use elevatorpro::utils::get_root_ip;
///
/// let ip = IpAddr::from_str("192.168.1.42").unwrap();
/// let root_ip = get_root_ip(ip);
/// assert_eq!(root_ip, "192.168.1");
/// ```
///
/// Returns a string containing the first three segments of the IP address.
pub fn get_root_ip(ip: IpAddr) -> String {
```

```
    match ip {
        IpAddr::V4(addr) => {
            let octets = addr.octets();
            format!("{}.{}.{}", octets[0], octets[1], octets[2])
        }
        IpAddr::V6(addr) => {
            let segments = addr.segments();
            let root_segments = &segments[..segments.len() - 1]; // Fjern siste segment
            root_segments.iter().map(|s| s.to_string()).collect::<Vec<_>>().join(":")
        }
    }
}
```

Innhald frå Rust-filer

Fil: elevator_pro_rebrand/src/lib.rs

```
#![warn(missing_docs)]
//! # This projects library
//!
//! This library manages configuration, network-communication between nodes, synchronization of world view across nodes and internally, elevator logic
//!
//! ## Overview
//! - **Config**: Handles configuration settings.
//! - **Utils**: Various helper functions.
//! - **Init**: System initialization.
//! - **Network**: Communication via UDP and TCP.
//! - **World View**: Managing and updating the world view.
//! - **Elevio**: Interface for elevator I/O.
//! - **Elevator Logic**: Task management and control logic for elevators.

/// Global variables
pub mod config;

/// Help functions
pub mod ip_help_functions;

/// Initialize functions
pub mod init;

/// Print functions with color coding
pub mod print;

/// Responsible for calculating cost and distribute tasks
pub mod manager;

/// Network communication via UDP and TCP.
pub mod network;

/// Management of the system's world view.
pub mod world_view;

/// Interface for elevator input/output. Only changes are documented here. For source code see: [https://github.com/TTK4145/driver-rust/tree/master/src/elevio]
pub mod elevio;

/// Elevator control logic and task handling.
pub mod elevator_logic;

/// Responsible for creating and running the backup-instnce
pub mod backup;
```

Innhald frå Rust-filer

Fil: elevator_pro_rebrand/src/config.rs

```
use std::net::Ipv4Addr;
use std::time::Duration;
use once_cell::sync::Lazy;
use std::sync::Mutex;

/// Network prefix: Initialized as the local network prefix in Sanntidshallen
pub static NETWORK_PREFIX: &str = "10.100.23";

/// Port for TCP between nodes
pub static PN_PORT: u16 = u16::MAX;
/// Port for TCP between node and local backup
pub static BCU_PORT: u16 = 50000;
/// Dummy port. Used for sending/recieving of UDP broadcasts
pub static DUMMY_PORT: u16 = 42069;

/// UDP broadcast listen address
pub static BC_LISTEN_ADDR: &str = "0.0.0.0";
/// UDP broadcast adress
pub static BC_ADDR: &str = "255.255.255.255";
/// Dummy IPv4 address when there is no internet connection (TODO: checking for internet could use an Option)
pub static OFFLINE_IP: Ipv4Addr = Ipv4Addr::new(69, 69, 69, 69);
/// IP to local elevator
pub static LOCAL_ELEV_IP: &str = "localhost:15657";

/// The default number of floors. Used for initializing the elevators in Sanntidshallen
pub const DEFAULT_NUM_FLOORS: u8 = 4;
/// Polling duration for reading from elevator
pub const ELEV_POLL: Duration = Duration::from_millis(25);

/// Error ID (TODO: Could use Some(ID) to identify errors)
pub const ERROR_ID: u8 = 255;

/// Index to ID of the master in a serialized worldview
pub const MASTER_IDX: usize = 1;
/// Key send in front of worldview on UDP broadcast, to filter out irrelevant broadcasts
pub const KEY_STR: &str = "Gruppe 25";

/// Timeout duration of TCP connections
pub const TCP_TIMEOUT: u64 = 5000; // i millisekunder
/// Probably unneccasary
pub const TCP_PER_U64: u64 = 10; // i millisekunder
/// Period between sending of UDP broadcasts
pub const UDP_PERIOD: Duration = Duration::from_millis(TCP_PER_U64);
/// Period between sending of TCP messages to master-node
pub const TCP_PERIOD: Duration = Duration::from_millis(TCP_PER_U64);
/// General period at 10 ms
pub const POLL_PERIOD: Duration = Duration::from_millis(10);
/// Period used to sleep before rechecking network status when you are offline
pub const OFFLINE_PERIOD: Duration = Duration::from_millis(100);
```

Innhald frå Rust-filer

```
/// Timeout duration of slave-nodes
pub const SLAVE_TIMEOUT: Duration = Duration::from_millis(100);

/// Timeout duration of master-nodes
pub const MASTER_TIMEOUT: Duration = Duration::from_secs(50); // 5 sekunder før failover

/// Timeout duration of backup-nodes
pub const BACKUP_TIMEOUT: Duration = Duration::from_secs(50); // 5 sekunder før failover

/// Size used for buffer when reading UDP broadcasts
pub const UDP_BUFFER: usize = u16::MAX as usize;

/// Time in seconds an elevator has to complete a task before its considered failed by master
pub const TASK_TIMEOUT: u64 = 100;




/// Bool to determine if program should print worldview
pub static PRINT_WV_ON: Lazy<Mutex<bool>> = Lazy::new(|| Mutex::new(true));
/// Bool to determine if program should print error's
pub static PRINT_ERR_ON: Lazy<Mutex<bool>> = Lazy::new(|| Mutex::new(true));
/// Bool to determine if program should print warnings
pub static PRINT_WARN_ON: Lazy<Mutex<bool>> = Lazy::new(|| Mutex::new(true));
/// Bool to determine if program should print ok-messages
pub static PRINT_OK_ON: Lazy<Mutex<bool>> = Lazy::new(|| Mutex::new(true));
/// Bool to determine if program should print info-messages
pub static PRINT_INFO_ON: Lazy<Mutex<bool>> = Lazy::new(|| Mutex::new(true));
/// Bool to determine if program should print other prints
pub static PRINT_ELSE_ON: Lazy<Mutex<bool>> = Lazy::new(|| Mutex::new(true));


/// Penalty for beeing busy
pub const BUSY_PENALTY: u32 = 5;
/// Penalty for going wrong direction
pub const WRONG_DIRECTION_PENALTY: u32 = 10;
```

Innhald frå Rust-filer

Fil: elevator_pro_rebrand/src/print.rs

```
use std::io::Write;
use termcolor::{Color, ColorChoice, ColorSpec, StandardStream, WriteColor};
use crate::{config, world_view::{Dirn, ElevatorBehaviour, serial}};
use ansi_term::Colour::{Green, Red, Yellow, Purple};

use unicode_width::UnicodeWidthStr;

/// Prints a message in a specified color to the terminal.
///
/// This function uses the `termcolor` crate to print a formatted message with
/// a given foreground color. If `PRINT_ELSE_ON` is `false`, the message will not be printed.
///
/// ## Parameters
/// - `msg`: The message to print.
/// - `color`: The color to use for the text output.
///
/// ## Example
/// ```
/// use termcolor::{Color, StandardStream, ColorSpec, WriteColor};
/// use elevatorpro::print;
///
/// print::color("Hello, World!".to_string(), Color::Green);
/// ```
///
/// **Note:** This function does not return a value and prints directly to the terminal.
/// If color output is not supported, the text may not appear as expected.
pub fn color(msg: String, color: Color) {
    let print_stat = config::PRINT_ELSE_ON.lock().unwrap().clone();

    if print_stat {
        let mut stdout = StandardStream::stdout(ColorChoice::Always);
        stdout.set_color(ColorSpec::new().set_fg(Some(color))).unwrap();
        writeln!(&mut stdout, "[CUSTOM]:  {}", msg).unwrap();
        stdout.set_color(&ColorSpec::new()).unwrap();
        println!("\r\n");
    }
}

/// Prints an error message in red to the terminal.
///
/// This function uses the `termcolor` crate to print an error message with a red foreground color.
/// If `PRINT_ERR_ON` is `false`, the message will not be printed.
///
/// ## Parameters
/// - `msg`: The error message to print.
///
/// ## Terminal output
/// - "\[ERROR\]:   {}", msg
///
/// ## Example
```

```
/// ```
/// use elevatorpro::print;
///
/// print::err("Something went wrong!".to_string());
/// print::err(format!("Something went wront: {}", e));
/// ```
///
/// **Note:** This function does not return a value and prints directly to the terminal.
/// If color output is not supported, the error message may not appear in red.
pub fn err(msg: String) {
    let print_stat = config::PRINT_ERR_ON.lock().unwrap().clone();

    if print_stat {
        let mut stdout = StandardStream::stdout(ColorChoice::Always);
        stdout.set_color(ColorSpec::new().set_fg(Some(Color::Red))).unwrap();
        writeln!(&mut stdout, "[ERROR]:   {}", msg).unwrap();
        stdout.set_color(&ColorSpec::new()).unwrap();
        println!("\r\n");
    }
}

/// Prints a warning message in yellow to the terminal.
///
/// This function uses the `termcolor` crate to print a warning message with a yellow foreground color.
/// If `PRINT_WARN_ON` is `false`, the message will not be printed.
///
/// ## Parameters
/// - `msg`: The warning message to print.
///
/// ## Terminal output
/// - "\[WARNING\]: {}", msg
///
/// ## Example
/// ```
/// use elevatorpro::print;
///
/// print::warn("This is a warning.".to_string());
/// ```
///
/// **Note:** This function does not return a value and prints directly to the terminal.
/// If color output is not supported, the warning message may not appear in yellow.
pub fn warn(msg: String) {
    let print_stat = config::PRINT_WARN_ON.lock().unwrap().clone();

    if print_stat {
        let mut stdout = StandardStream::stdout(ColorChoice::Always);
        stdout.set_color(ColorSpec::new().set_fg(Some(Color::Yellow))).unwrap();
        writeln!(&mut stdout, "[WARNING]: {}", msg).unwrap();
        stdout.set_color(&ColorSpec::new()).unwrap();
        println!("\r\n");
    }
}
```

Innhald frå Rust-filer

```rust
/// Prints a success message in green to the terminal.
///
/// This function uses the `termcolor` crate to print a success message with a green foreground color.
/// If `PRINT_OK_ON` is `false`, the message will not be printed.
///
/// ## Parameters
/// - `msg`: The success message to print.
///
/// ## Terminal output
/// - "\[OK\]:      {}", msg
///
/// ## Example
/// ```
/// use elevatorpro::print;
///
/// print::ok("Operation successful.".to_string());
/// ```
///
/// **Note:** This function does not return a value and prints directly to the terminal.
/// If color output is not supported, the success message may not appear in green.
pub fn ok(msg: String) {
    let print_stat = config::PRINT_OK_ON.lock().unwrap().clone();

    if print_stat {
        let mut stdout = StandardStream::stdout(ColorChoice::Always);
        stdout.set_color(ColorSpec::new().set_fg(Some(Color::Green))).unwrap();
        writeln!(&mut stdout, "[OK]:      {}", msg).unwrap();
        stdout.set_color(&ColorSpec::new()).unwrap();
        println!("\r\n");
    }
}

/// Prints an informational message in light blue to the terminal.
///
/// This function uses the `termcolor` crate to print an informational message with a light blue foreground color.
/// If `PRINT_INFO_ON` is `false`, the message will not be printed.
///
/// ## Parameters
/// - `msg`: The informational message to print.
///
/// ## Terminal output
/// - "\[INFO\]:    {}", msg
///
/// ## Example
/// ```
/// use elevatorpro::print;
///
/// print::info("This is an informational message.".to_string());
/// ```
///
/// **Note:** This function does not return a value and prints directly to the terminal.
```

```
/// If color output is not supported, the informational message may not appear in light blue.
pub fn info(msg: String) {
    let print_stat = config::PRINT_INFO_ON.lock().unwrap().clone();

    if print_stat {
        let mut stdout = StandardStream::stdout(ColorChoice::Always);
        stdout.set_color(ColorSpec::new().set_fg(Some(Color::Rgb(102, 178, 255/*lyseblå*/)))).unwrap();
        writeln!(&mut stdout, "[INFO]:    {}", msg).unwrap();
        stdout.set_color(&ColorSpec::new()).unwrap();
        println!("\r\n");
    }
}

/// Prints a master-specific message in pink to the terminal.
///
/// This function uses the `termcolor` crate to print a master-specific message with a pink foreground color.
/// If `PRINT_ELSE_ON` is `false`, the message will not be printed.
///
/// ## Parameters
/// - `msg`: The master-specific message to print.
///
/// ## Terminal output
/// - "\[MASTER\]:  {}", msg
///
/// ## Example
/// ```
/// use elevatorpro::print;
///
/// print::master("Master process initialized.".to_string());
/// ```
///
/// **Note:** This function does not return a value and prints directly to the terminal.
/// If color output is not supported, the master message may not appear in pink.
pub fn master(msg: String) {
    let print_stat = config::PRINT_ELSE_ON.lock().unwrap().clone();

    if print_stat {
        let mut stdout = StandardStream::stdout(ColorChoice::Always);
        stdout.set_color(ColorSpec::new().set_fg(Some(Color::Rgb(255, 51, 255/*Rosa*/)))).unwrap();
        writeln!(&mut stdout, "[MASTER]:  {}", msg).unwrap();
        stdout.set_color(&ColorSpec::new()).unwrap();
        println!("\r\n");
    }
}

/// Prints a slave-specific message in orange to the terminal.
///
/// This function uses the `termcolor` crate to print a slave-specific message with an orange foreground color.
/// If `PRINT_ELSE_ON` is `false`, the message will not be printed.
///
/// ## Parameters
/// - `msg`: The slave-specific message to print.
```

```
///
/// ## Terminal output
/// - "\[SLAVE\]:   {}", msg
///
/// ## Example
/// ```
/// use elevatorpro::print;
///
/// print::slave("Slave process running.".to_string());
/// ```
///
/// **Note:** This function does not return a value and prints directly to the terminal.
/// If color output is not supported, the slave message may not appear in orange.
pub fn slave(msg: String) {
    let print_stat = config::PRINT_ELSE_ON.lock().unwrap().clone();

    if print_stat {
        let mut stdout = StandardStream::stdout(ColorChoice::Always);
        stdout.set_color(ColorSpec::new().set_fg(Some(Color::Rgb(153, 76, 0/*Tilfeldig*/)))).unwrap();
        writeln!(&mut stdout, "[SLAVE]:   {}", msg).unwrap();
        stdout.set_color(&ColorSpec::new()).unwrap();
        println!("\r\n");
    }
}

/// Prints an error message with a cosmic twist, displaying the message in a rainbow of colors.
///
/// This function prints a message when something happens that is theoretically impossible,
/// such as a "cosmic ray flipping a bit" scenario. It starts with a red "[ERROR]:" label and
/// follows with the rest of the message displayed in a rainbow pattern.
///
/// # Parameters
/// - `fun`: The function name or description of the issue that led to this cosmic error.
///
/// ## Terminal output
/// - "[ERROR]: Cosmic rays flipped a bit!    1 0 IN: {fun}"
///    Where `{fun}` is replaced by the provided `fun` parameter, and the rest of the message is displayed in rainbow colors.
///
/// # Example
/// ```
/// use elevatorpro::print;
///
/// print::cosmic_err("Something impossible happened".to_string());
/// ```
///
/// **Note:** This function does not return a value and prints directly to the terminal. The message will be printed in a rainbow of colors.
pub fn cosmic_err(fun: String) {
    let mut stdout = StandardStream::stdout(ColorChoice::Always);
    // Skriv ut "[ERROR]:" i rød
    stdout.set_color(ColorSpec::new().set_fg(Some(Color::Red))).unwrap();
```

```
    write!(&mut stdout, "[ERROR]: ").unwrap();
    // Definer regnbuefargene
    let colors = [
        Color::Red,
        Color::Yellow,
        Color::Green,
        Color::Cyan,
        Color::Blue,
        Color::Magenta,
    ];
    // Resten av meldingen i regnbuefarger
    let message = format!("Cosmic rays flipped a bit!    1 0  IN: {}", fun);
    for (i, c) in message.chars().enumerate() {
        let color = colors[i % colors.len()];
        stdout.set_color(ColorSpec::new().set_fg(Some(color))).unwrap();
        write!(&mut stdout, "{}", c).unwrap();
    }
    // Tilbakestill fargen
    stdout.set_color(&ColorSpec::new()).unwrap();
    println!();
}

/// Hjelpefunksjon for å sikre at kolonner har fast breidde
fn pad_text(text: &str, width: usize) -> String {
    let visible_width = UnicodeWidthStr::width(text);
    let padding = width.saturating_sub(visible_width);
    format!("{}{}", text, " ".repeat(padding))
}

/// Logger `wv` i eit fint tabellformat
pub fn worldview(worldview: Vec<u8>) {
    let print_stat = config::PRINT_WV_ON.lock().unwrap().clone();
    if !print_stat {
        return;
    }

    let wv_deser = serial::deserialize_worldview(&worldview);

    // Overskrift
    println!("{}", Purple.bold().paint(""));
    println!("{}", Purple.bold().paint("        WORLD VIEW STATUS        "));
    println!("{}", Purple.bold().paint(""));

    // Generell info-tabell
    println!("");
    println!("{}", ansi_term::Colour::White.bold().paint(" Num heiser   MasterID  Pending tasks       "));
    println!("");

    println!(
        " {:<11}  {:<8}                      ",
        wv_deser.get_num_elev(),
        wv_deser.master_id
```

```
);

for (floor, calls) in wv_deser.hall_request.iter().enumerate().rev() {
    println!(
        " floor:{:<5}           {} {}              ",
        floor,
        if calls[1] { "" } else { "" }, // Ned
        if calls[0] { "" } else { "" }  // Opp
    );
}

println!("");

// Heisstatus-tabell
println!("");
println!("{}", ansi_term::Colour::White.bold().paint(" ID    Dør       Obstruksjon   Tasks         Siste etasje Calls (Etg:Call)
Elev status   "));
println!("");

for elev in &wv_deser.elevator_containers {
    let id_text = pad_text(&format!("{}", elev.elevator_id), 4);
    let door_text = if elev.behaviour == ElevatorBehaviour::DoorOpen {
        pad_text(&Yellow.paint("Open").to_string(), 17)
    } else {
        pad_text(&Green.paint("Lukka").to_string(), 17)
    };
    let obstruction_text = if elev.obstruction {
        pad_text(&Red.paint("Ja").to_string(), 21)
    } else {
        pad_text(&Green.paint("Nei").to_string(), 21)
    };

    let tasks_emoji: Vec<String> = elev.cab_requests.iter().enumerate().rev()
        .map(|(floor, task)| format!("{:<2} {}", floor, if *task { "" } else { "" }))
        .collect();

    let call_list_emoji: Vec<String> = elev.tasks.iter().enumerate().rev()
        .map(|(floor, calls)| format!("{:<2} {} {}", floor, if calls[1] { "" } else { "" }, if calls[0] { "" } else { "" }))
        .collect();

    let task_status = match (elev.dirn, elev.behaviour) {
        (_, ElevatorBehaviour::Idle) => pad_text(&Green.paint("Idle").to_string(), 22),
        (Dirn::Up, ElevatorBehaviour::Moving) => pad_text(&Yellow.paint("   Moving").to_string(), 23),
        (Dirn::Down, ElevatorBehaviour::Moving) => pad_text(&Yellow.paint("   Moving").to_string(), 23),
        (Dirn::Stop, ElevatorBehaviour::Moving) => pad_text(&Yellow.paint("Not Moving").to_string(), 22),
        (_, ElevatorBehaviour::DoorOpen) => pad_text(&Purple.paint("Door Open").to_string(), 22),
        (_, ElevatorBehaviour::Error) => pad_text(&Red.paint("Error").to_string(), 22),
    };

    let max_rows = std::cmp::max(tasks_emoji.len(), call_list_emoji.len());

    for i in 0..max_rows {
            let task_entry = tasks_emoji.get(i).cloned().unwrap_or_else(|| "  ".to_string());
            let call_entry = call_list_emoji.get(i).cloned().unwrap_or_else(|| "  ".to_string());

            if i == 0 {
                println!(
                    " {}  {}  {}  {:<11}  {:<11}  {:<18}  {} ",
                    id_text, door_text, obstruction_text, task_entry, elev.last_floor_sensor, call_entry, task_status
                );
            } else {
                println!(
                    "                              {:<11}              {:<18}                ",
                    task_entry, call_entry
                );
            }
        }

        println!("");
    }

    println!("");
}
```

Innhald frå Rust-filer

Fil: elevator_pro_rebrand/src/backup/mod.rs

```
use std::env;
use std::net::SocketAddr;
use std::process::Command;
use std::sync::atomic::{AtomicBool, Ordering};
use std::io::{self, Write};
use socket2::{Socket, Domain, Type, Protocol};
use tokio::net::{TcpListener, TcpStream};
use tokio::io::{AsyncReadExt, AsyncWriteExt};
use tokio::sync::watch;
use tokio::time::{sleep, Duration, timeout};

// Tilpass desse importane til prosjektet ditt:
use crate::{config, init, ip_help_functions, world_view};
use crate::print;
use crate::network::local_network;

// Global variabel for å sjå om backup-terminalen allereie er starta
static BACKUP_STARTED: AtomicBool = AtomicBool::new(false);

/// Opprett ein gjennbrukbar TcpListener med reuse_address aktivert.
pub fn create_reusable_listener(port: u16) -> TcpListener {
    let addr: SocketAddr = format!("0.0.0.0:{}", port)
        .parse()
        .expect("Ugyldig adresse");
    let socket = Socket::new(Domain::IPV4, Type::STREAM, Some(Protocol::TCP))
        .expect("Klarte ikkje opprette socket");
    socket.set_nonblocking(true).expect("msg");
    socket.set_reuse_address(true)
        .expect("Klarte ikkje setje reuse_address");
    socket.bind(&addr.into())
        .expect("Klarte ikkje binde socketen");
    socket.listen(128)
        .expect("Klarte ikkje lytte på socketen");
    TcpListener::from_std(socket.into())
        .expect("Klarte ikkje opprette TcpListener")
}

/// Startar backup-terminalen i eit nytt terminalvindu  berre om han ikkje allereie er starta.
fn start_backup_terminal() {
    if !BACKUP_STARTED.load(Ordering::SeqCst) {
        let current_exe = env::current_exe().expect("Klarte ikkje hente ut den kjørbare fila");
        // Eksempel med gnome-terminal og --geometry for å spesifisere vindaugets storleik.
        let _child = Command::new("gnome-terminal")
            .arg("--geometry=400x24")
            .arg("--")
            .arg(current_exe.to_str().unwrap())
            .arg("backup")
            .spawn()
            .expect("Feil ved å starte backupterminalen");
        BACKUP_STARTED.store(true, Ordering::SeqCst);
    }
}

/// Handterar backup-klientar: Sender ut worldview kontinuerleg.
async fn handle_backup_client(mut stream: TcpStream, rx: watch::Receiver<Vec<u8>>) {
    loop {
        let wv = rx.borrow().clone();
        if let Err(e) = stream.write_all(&wv).await {
            eprintln!("Backup send error: {}", e);
            // Set BACKUP_STARTED til false, slik at ein ny backup-terminal kan startast
            BACKUP_STARTED.store(false, Ordering::SeqCst);
            start_backup_terminal();
            // Avslutt løkka for denne klienten for å unngå evig loop.
            break;
        }
        sleep(Duration::from_millis(1000)).await;
    }
}

// Backup-serveren: Lytter på tilkoplingar frå backup-klientar og sender ut den nyaste worldview.
/// Function to start and maintain connection to the backup-program
///
/// ## Parameters
/// `wv_watch_rx`: Rx on watch the worldview is being sent on in the system
///
/// ## Behavior
/// - Sets up a reusable TCP listener and starts a backup program in a new terminal
/// - Continously sends the latest worldview to the backup asynchronously
/// - Continously reads the latest worldview shich will be sent
///
/// ## Note
/// This function is permanently blocking, and should be ran asynchronously
pub async fn start_backup_server(wv_watch_rx: watch::Receiver<Vec<u8>>) {
    println!("Backup-serveren startar...");

    // Bruk ein gjennbrukbar listener.
    let listener = create_reusable_listener(config::BCU_PORT);
    let wv = world_view::get_wv(wv_watch_rx.clone());
    let (tx, rx) = watch::channel(wv.clone());

    // Start backup-terminalen éin gong.
    start_backup_terminal();

    // Task for å handtere backup-klientar.
    tokio::spawn(async move {
        loop {
            let (socket, _) = listener
                .accept()
                .await
                .expect("Klarte ikkje akseptere backup-kopling");
            handle_backup_client(socket, rx.clone()).await;
        }
    });

    // Oppdater kontinuerleg worldview til backup-klientane.
    loop {
        let new_wv = world_view::get_wv(wv_watch_rx.clone());
        tx.send(new_wv).expect("Klarte ikkje sende til backup-klientane");
        sleep(Duration::from_millis(1000)).await;
    }
}

/// Backup-klienten: Koplar seg til backup-serveren, les data kontinuerleg og skriv ut worldview.
pub async fn run_as_backup() -> Option<world_view::ElevatorContainer> {
    println!("Starter backup-klient...");
    let mut current_wv = init::initialize_worldview(None).await;
    let mut retries = 0;

    loop {
        match timeout(
            config::MASTER_TIMEOUT,
            TcpStream::connect(format!("127.0.0.1:{}", config::BCU_PORT))
        ).await {
            Ok(Ok(mut stream)) => {
                retries = 0;
                let mut buf = vec![0u8; 1024];
                // Les data i ein løkke for kontinuerleg oppdatering
                loop {
                    match stream.read(&mut buf).await {
                        Ok(0) => {
                            eprintln!("Master koplinga vart avslutta.");
                            break;
                        },
                        Ok(n) => {
                            current_wv = buf[..n].to_vec();
                            // Rydd skjermen og sett markøren øvst
                            print!("\x1B[2J\x1B[H");

                            // Sørg for at utskrifta skjer umiddelbart
                            io::stdout().flush().unwrap();

                            print::worldview(current_wv.clone());
                        },
                        Err(e) => {
                            eprintln!("Lesefeil frå master: {}", e);
                            break;
                        }
                    }
                    sleep(Duration::from_millis(500)).await;
                }
            },
            _ => {
                retries += 1;
                eprintln!("Kunne ikkje koble til master, retry {}.", retries);
                if retries > 50 {
                    eprintln!("Master feila, promoterer backup til master!");
                    // Her kan failover-logikken setjast i gang, t.d. køyre master-logikken.
                    match world_view::extract_self_elevator_container(current_wv) {
                        Some(container) => return Some(container),
                        None => {
                            print::warn(format!("Failed to extract self elevator container"));
                            return None;
                        }
                    }

                }
            }
        }
        sleep(Duration::from_secs(1)).await;
    }
}
```

Innhald frå Rust-filer

Fil: elevator_pro_rebrand/src/elevio/poll.rs

```
//! Listens for events from the elevator.

use crossbeam_channel as cbc;
use std::sync::atomic::Ordering;
use std::thread;
use std::time;

use crate::network;
use crate::elevio::{CallButton, CallType, elev};


#[doc(hidden)]
pub fn call_buttons(elev: elev::Elevator, ch: cbc::Sender<CallButton>, period: time::Duration) {
    let mut prev = vec![[false; 3]; elev.num_floors.into()];
    loop {
        for f in 0..elev.num_floors {
            for c in 0..3 {
                let v = elev.call_button(f, c);
                if v && prev[f as usize][c as usize] != v {
                    println!("{:?}",c);
                    ch.send(CallButton { floor: f, call_type: CallType::from(c), elev_id: network::read_self_id()}).unwrap();
                }
                prev[f as usize][c as usize] = v;
            }
        }
        thread::sleep(period)
    }
}

#[doc(hidden)]
pub fn floor_sensor(elev: elev::Elevator, ch: cbc::Sender<u8>, period: time::Duration) {
    let mut prev = u8::MAX;
    loop {
        if let Some(f) = elev.floor_sensor() {
            if f != prev {
                ch.send(f).unwrap();
                prev = f;
            }
        }
        thread::sleep(period)
    }
}

#[doc(hidden)]
pub fn stop_button(elev: elev::Elevator, ch: cbc::Sender<bool>, period: time::Duration) {
    let mut prev = false;
    loop {
        let v = elev.obstruction();
        if prev != v {
            ch.send(v).unwrap();
            prev = v;
        }
        thread::sleep(period)
    }
}

#[doc(hidden)]
pub fn obstruction(elev: elev::Elevator, ch: cbc::Sender<bool>, period: time::Duration) {
    let mut prev = false;
    loop {
        let v = elev.stop_button();
        if prev != v {
            ch.send(v).unwrap();
            prev = v;
        }
        thread::sleep(period)
    }
}
```

Innhald frå Rust-filer

Fil: elevator_pro_rebrand/src/elevio/elev.rs

```
#![allow(dead_code)]
use std::fmt;
use std::io::*;
use std::net::TcpStream;
use std::sync::*;

#[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 {
    pub fn init(addr: &str, num_floors: u8) -> Result<Elevator> {
        Ok(Self {
            socket: Arc::new(Mutex::new(TcpStream::connect(addr)?)),
            num_floors,
        })
    }

    pub fn motor_direction(&self, dirn: u8) {
        let buf = [1, dirn, 0, 0];
        let mut sock = self.socket.lock().unwrap();
        sock.write(&buf).unwrap();
    }

    pub fn call_button_light(&self, floor: u8, call: u8, on: bool) {
        let buf = [2, call, floor, on as u8];
        let mut sock = self.socket.lock().unwrap();
        sock.write(&buf).unwrap();
    }

    pub fn floor_indicator(&self, floor: u8) {
        let buf = [3, floor, 0, 0];
        let mut sock = self.socket.lock().unwrap();
        sock.write(&buf).unwrap();
    }

    pub fn door_light(&self, on: bool) {
        let buf = [4, on as u8, 0, 0];
        let mut sock = self.socket.lock().unwrap();
        sock.write(&buf).unwrap();
    }

    pub fn stop_button_light(&self, on: bool) {
        let buf = [5, on as u8, 0, 0];
        let mut sock = self.socket.lock().unwrap();
        sock.write(&buf).unwrap();
    }

    pub fn call_button(&self, floor: u8, call: u8) -> bool {
        let mut buf = [6, call, floor, 0];
        let mut sock = self.socket.lock().unwrap();
        sock.write(&mut buf).unwrap();
        sock.read(&mut buf).unwrap();
        buf[1] != 0
    }

    pub fn floor_sensor(&self) -> Option<u8> {
        let mut buf = [7, 0, 0, 0];
        let mut sock = self.socket.lock().unwrap();
        sock.write(&buf).unwrap();
        sock.read(&mut buf).unwrap();
        if buf[1] != 0 {
            Some(buf[2])
        } else {
            None
        }
    }

    pub fn stop_button(&self) -> bool {
        let mut buf = [8, 0, 0, 0];
        let mut sock = self.socket.lock().unwrap();
        sock.write(&buf).unwrap();
        sock.read(&mut buf).unwrap();
        buf[1] != 0
    }

    pub fn obstruction(&self) -> bool {
        let mut buf = [9, 0, 0, 0];
        let mut sock = self.socket.lock().unwrap();
        sock.write(&buf).unwrap();
        sock.read(&mut buf).unwrap();
        buf[1] != 0
    }
}

impl fmt::Display for Elevator {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        let addr = self.socket.lock().unwrap().peer_addr().unwrap();
        write!(f, "Elevator@{}({})", addr, self.num_floors)
    }
}
```

Innhald frå Rust-filer

Fil: elevator_pro_rebrand/src/elevio/mod.rs

```
#[doc(hidden)]
pub mod elev;
pub mod poll;

use crate::print;
use crate::config;

use serde::{Serialize, Deserialize};
use std::hash::{Hash, Hasher};


/// Represents different types of elevator messages.
#[derive(Debug, Clone, PartialEq, Eq)]
pub enum ElevMsgType {
    /// Call button press event.
    CALLBTN,
    /// Floor sensor event.
    FLOORSENS,
    /// Stop button press event.
    STOPBTN,
    /// Obstruction detected event.
    OBSTRX,
}

/// Represents a message related to elevator events.
#[derive(Debug, Clone)]
pub struct ElevMessage {
    /// The type of elevator message.
    pub msg_type: ElevMsgType,
    /// Optional call button information, if applicable.
    pub call_button: Option<CallButton>,
    /// Optional floor sensor reading, indicating the current floor.
    pub floor_sensor: Option<u8>,
    /// Optional stop button state (`true` if pressed).
    pub stop_button: Option<bool>,
    /// Optional obstruction status (`true` if obstruction detected).
    pub obstruction: Option<bool>,
}

/// Represents the type of call for an elevator.
///
/// This enum is used to differentiate between different types of elevator requests.
///
/// ## Variants
/// - `UP`: A request to go up.
/// - `DOWN`: A request to go down.
/// - `INSIDE`: A request made from inside the elevator.
/// - `COSMIC_ERROR`: An invalid call type (used as an error fallback).
#[derive(Serialize, Deserialize, Debug, Clone, Copy, PartialEq, Eq, Hash)]
```

```
#[repr(u8)] // Ensures the enum is stored as a single byte.
#[allow(non_camel_case_types)]
pub enum CallType {
    /// Call to go up.
    UP = 0,

    /// Call to go down.
    DOWN = 1,

    /// Call from inside the elevator.
    INSIDE = 2,

    /// Represents an invalid call type.
    COSMIC_ERROR = 255,
}
impl From<u8> for CallType {
    /// Converts a `u8` value into a `CallType`.
    ///
    /// If the value does not match a valid `CallType`, it logs an error and returns `COSMIC_ERROR`.
    ///
    /// # Examples
    /// ```
    /// # use elevatorpro::elevio::poll::CallType;
    ///
    /// let call_type = CallType::from(0);
    /// assert_eq!(call_type, CallType::UP);
    ///
    /// let invalid_call = CallType::from(10);
    /// assert_eq!(invalid_call, CallType::COSMIC_ERROR);
    /// ```
    fn from(value: u8) -> Self {
        match value {
            0 => CallType::UP,
            1 => CallType::DOWN,
            2 => CallType::INSIDE,
            _ => {
                print::cosmic_err("Call type does not exist".to_string());
                CallType::COSMIC_ERROR
            },
        }
    }
}

/// Represents a button press in an elevator system.
///
/// Each button press consists of:
/// - `floor`: The floor where the button was pressed.
/// - `call`: The type of call (up, down, inside).
/// - `elev_id`: The ID of the elevator (relevant for `INSIDE` calls).
#[derive(Serialize, Deserialize, Debug, Clone, Copy, Eq)]
pub struct CallButton {
    /// The floor where the call was made.
```

```
    pub floor: u8,

    /// The type of call (UP, DOWN, or INSIDE).
    pub call_type: CallType,

    /// The ID of the elevator making the call (only relevant for `INSIDE` calls).
    pub elev_id: u8,
}
impl Default for CallButton {
    fn default() -> Self {
        CallButton{floor: 1, call_type: CallType::INSIDE, elev_id: config::ERROR_ID}
    }
}


impl PartialEq for CallButton {
    /// Custom equality comparison for `CallButton`.
    ///
    /// Two call buttons are considered equal if they have the same floor and call type.
    /// However, for `INSIDE` calls, the `elev_id` must also match.
    ///
    /// # Examples
    /// ```
    /// # use elevatorpro::elevio::poll::{CallType, CallButton};
    ///
    /// let button1 = CallButton { floor: 3, call: CallType::UP, elev_id: 1 };
    /// let button2 = CallButton { floor: 3, call: CallType::UP, elev_id: 2 };
    ///
    /// assert_eq!(button1, button2); // Same floor & call type
    ///
    /// let inside_button1 = CallButton { floor: 2, call: CallType::INSIDE, elev_id: 1 };
    /// let inside_button2 = CallButton { floor: 2, call: CallType::INSIDE, elev_id: 2 };
    ///
    /// assert_ne!(inside_button1, inside_button2); // Different elevators
    /// ```
    fn eq(&self, other: &Self) -> bool {
        // Hvis call er INSIDE, sammenligner vi også elev_id
        if self.call_type == CallType::INSIDE {
            self.floor == other.floor && self.call_type == other.call_type && self.elev_id == other.elev_id
        } else {
            // For andre CallType er det tilstrekkelig å sammenligne floor og call
            self.floor == other.floor && self.call_type == other.call_type
        }
    }
}
impl Hash for CallButton {
    /// Custom hashing function to ensure consistency with `PartialEq`.
    ///
    /// This ensures that buttons with the same floor and call type have the same hash.
    /// For `INSIDE` calls, the elevator ID is also included in the hash.
    fn hash<H: Hasher>(&self, state: &mut H) {
        // Sørger for at hash er konsistent med eq
```

```
        self.floor.hash(state);
        self.call_type.hash(state);
        if self.call_type == CallType::INSIDE {
            self.elev_id.hash(state);
        }
    }
}
```

Innhald frå Rust-filer

Fil: elevator_pro_rebrand/src/elevator_logic/lights.rs

```
use tokio::sync::watch;

use crate::{elevio::elev::Elevator, world_view::{self, ElevatorContainer}};


/// Sets all hall lights
///
/// ## Parameters
/// `wv`: Serialized worldview
/// `e`: Elevator instance
///
/// ## Behavior:
/// The function goes through all hall requests in the worldview, and sets hall lights if the corresponding lights on/off based on the boolean value in the worldview.
/// The function skips any hall lights on floors grater than the elevators num_floors, as well as down on floor nr. 0 and up on floor nr. e.num_floors
///
/// ## Note
/// The function only sets the lights once per call, and needs to be recalled every time the lights needs to be updated
pub fn set_hall_lights(wv: Vec<u8>, e: Elevator, container: &ElevatorContainer) {
    let wv_deser = world_view::serial::deserialize_worldview(&wv);

    for (i, [up, down]) in wv_deser.hall_request.iter().enumerate() {
        let floor = i as u8;
        if floor > e.num_floors {
            break;
        }

        if floor != 0 {
            e.call_button_light(floor, 1, *down);
        }
        if floor != e.num_floors {
            e.call_button_light(floor, 0, *up);
        }
    }
}

/// The function sets the cab light on last_floor_sensor
pub fn set_cab_light(e: Elevator, last_floor: u8) {
    e.floor_indicator(last_floor);
}

/// The function sets the door open light on
pub fn set_door_open_light(e: Elevator) {
    e.door_light(true);
}

/// The function sets the door open light off
pub fn clear_door_open_light(e: Elevator) {
    e.door_light(false);
}

/// The function sets the stop button light on
pub fn set_stop_button_light(e: Elevator) {
    e.stop_button_light(true);
}

/// The function sets the stop button light off
pub fn clear_stop_button_light(e: Elevator) {
    e.stop_button_light(false);
}
```

Innhald frå Rust-filer

Fil: elevator_pro_rebrand/src/elevator_logic/request.rs

```
use std::u8::{self, MIN};

use crate::{elevio::elev, print, world_view::{Dirn, ElevatorBehaviour, ElevatorContainer}};

#[derive(Debug, Clone, Copy)]
pub struct DirnBehaviourPair {
    pub dirn: Dirn,
    pub behaviour: ElevatorBehaviour,
}

/////Requests
fn above(elevator: &ElevatorContainer) -> bool {
    for floor in (elevator.last_floor_sensor as usize + 1)..elevator.tasks.len() {
        for btn in 0..2 {
            if elevator.tasks[floor][btn] {
                return true;
            }
        }
        if elevator.cab_requests[floor] {
            return true;
        }
    }
    false
}

fn inside_above(elevator: &ElevatorContainer) -> bool {
    for floor in (elevator.last_floor_sensor as usize + 1)..elevator.tasks.len() {
        if elevator.cab_requests[floor] {
            return true;
        }
    }
    false
}

fn below(elevator: &ElevatorContainer) -> bool {
    for floor in 0..elevator.last_floor_sensor as usize {
        for btn in 0..2 {
            if elevator.tasks[floor][btn] {
                return true;
            }
        }
        if elevator.cab_requests[floor] {
            return true;
        }
    }
    false
}

fn insie_below(elevator: &ElevatorContainer) -> bool {
    for floor in 0..elevator.last_floor_sensor as usize {
        if elevator.cab_requests[floor] {
            return true;
        }
    }
    false
}

fn here(elevator: &ElevatorContainer) -> bool {
    // if elevator.last_floor_sensor >= elevator.num_floors{
    //     return false; // retuner ved feil
    // }
    for btn in 0..2 {
        if elevator.tasks[elevator.last_floor_sensor as usize][btn] {
            return true;
        }
    }
    if elevator.cab_requests[elevator.last_floor_sensor as usize] {
        return true;
    }
    false
}

fn get_here_dirn(elevator: &ElevatorContainer) -> Dirn {
    if elevator.tasks[elevator.last_floor_sensor as usize][0] {
        return Dirn::Up;
    } else if elevator.tasks[elevator.last_floor_sensor as usize][1] {
        return Dirn::Down;
    } else {
        return Dirn::Stop;
    }

}


pub fn moving_towards_cab_call(elevator: &ElevatorContainer) -> bool {
    match elevator.dirn {
        Dirn::Up => {
            return inside_above(&elevator.clone());
        },
        Dirn::Down => {
            return insie_below(&elevator.clone());
        },
        Dirn::Stop => {
            return false;
        }
    }
}


pub fn choose_direction(elevator: &ElevatorContainer) -> DirnBehaviourPair {
    match elevator.dirn {
        Dirn::Up => {
            if above(elevator) {
                DirnBehaviourPair { dirn: Dirn::Up, behaviour: ElevatorBehaviour::Moving }
            } else if here(elevator) {
                DirnBehaviourPair { dirn: Dirn::Down, behaviour: ElevatorBehaviour::DoorOpen }
            } else if below(elevator) {
                DirnBehaviourPair { dirn: Dirn::Down, behaviour: ElevatorBehaviour::Moving }
            } else {
                DirnBehaviourPair { dirn: Dirn::Stop, behaviour: ElevatorBehaviour::Idle }
            }
        }
        Dirn::Down => {
            if below(elevator) {
                DirnBehaviourPair { dirn: Dirn::Down, behaviour: ElevatorBehaviour::Moving }
            } else if here(elevator) {
                DirnBehaviourPair { dirn: Dirn::Up, behaviour: ElevatorBehaviour::DoorOpen }
            } else if above(elevator) {
                DirnBehaviourPair { dirn: Dirn::Up, behaviour: ElevatorBehaviour::Moving }
            } else {
                DirnBehaviourPair { dirn: Dirn::Stop, behaviour: ElevatorBehaviour::Idle }
            }
        }
        Dirn::Stop => {
            if here(elevator) {
                DirnBehaviourPair { dirn: get_here_dirn(elevator), behaviour: ElevatorBehaviour::DoorOpen }
            } else if above(elevator) {
                DirnBehaviourPair { dirn: Dirn::Up, behaviour: ElevatorBehaviour::Moving }
            } else if below(elevator) {
                DirnBehaviourPair { dirn: Dirn::Down, behaviour: ElevatorBehaviour::Moving }
            } else {
                // print::err(4.to_string());
                DirnBehaviourPair { dirn: Dirn::Stop, behaviour: ElevatorBehaviour::Idle }
            }
        }
    }
}

pub fn should_stop(elevator: &ElevatorContainer) -> bool {
    let floor = elevator.last_floor_sensor as usize;

    if elevator.cab_requests[floor] {
        return true;
    }

    match elevator.dirn {
        Dirn::Down => {
            elevator.tasks[floor][1] || !below(elevator)
        }
        Dirn::Up => {
            elevator.tasks[floor][0] || !above(elevator)
        }
        Dirn::Stop => true,
    }
```

```
}

pub fn was_outside(elevator: &ElevatorContainer) -> bool {
    let floor = elevator.last_floor_sensor as usize;

    match elevator.dirn {
        Dirn::Down => {
            elevator.tasks[floor][1] || !below(elevator)
        }
        Dirn::Up => {
            elevator.tasks[floor][0] || !above(elevator)
        }
        Dirn::Stop => true,
    }
}


pub fn clear_at_current_floor(elevator: &mut ElevatorContainer) {
    match elevator.dirn {
        Dirn::Up => {
            elevator.cab_requests[elevator.last_floor_sensor as usize] = false;
            // Master clearer hall_request
        },
        Dirn::Down => {
            elevator.cab_requests[elevator.last_floor_sensor as usize] = false;
            // Master clearer hall_request
        },
        Dirn::Stop => {
            elevator.cab_requests[elevator.last_floor_sensor as usize] = false;
        },
    }
}
```

Innhald frå Rust-filer

Fil: elevator_pro_rebrand/src/elevator_logic/self_elevator.rs

```rust
use tokio::task::yield_now;
use tokio::time::{sleep, Duration};
use crossbeam_channel as cbc;
use tokio::process::Command;
use std::sync::atomic::Ordering;
use tokio::sync::{mpsc, watch};

use crate::network;
use crate::world_view::ElevatorContainer;
use crate::{world_view::{Dirn, ElevatorBehaviour}, network::local_network, config, print, elevio, elevio::elev as e};

use super::timer::Timer;



struct LocalElevTxs {
    call_button: cbc::Sender<elevio::CallButton>,
    floor_sensor: cbc::Sender<u8>,
    stop_button: cbc::Sender<bool>,
    obstruction: cbc::Sender<bool>,
}

struct LocalElevRxs {
    call_button: cbc::Receiver<elevio::CallButton>,
    floor_sensor: cbc::Receiver<u8>,
    stop_button: cbc::Receiver<bool>,
    obstruction: cbc::Receiver<bool>,
}

struct LocalElevChannels {
    pub rxs: LocalElevRxs,
    pub txs: LocalElevTxs,
}

impl LocalElevChannels {
    pub fn new() -> Self {
        let (call_button_tx, call_button_rx) = cbc::unbounded::<elevio::CallButton>();
        let (floor_sensor_tx, floor_sensor_rx) = cbc::unbounded::<u8>();
        let (stop_button_tx, stop_button_rx) = cbc::unbounded::<bool>();
        let (obstruction_tx, obstruction_rx) = cbc::unbounded::<bool>();

        LocalElevChannels {
            rxs: LocalElevRxs { call_button: call_button_rx, floor_sensor: floor_sensor_rx, stop_button: stop_button_rx, obstruction: obstruction_rx },
            txs: LocalElevTxs { call_button: call_button_tx, floor_sensor: floor_sensor_tx, stop_button: stop_button_tx, obstruction: obstruction_tx }
        }
    }
}
```

```
/// ### Henter ut lokal IP adresse
fn get_ip_address() -> String {
    let self_id = network::read_self_id();
    format!("{}.{}", config::NETWORK_PREFIX, self_id)
}

/// ### Starter elevator_server
///
/// Tar høyde for om du er på windows eller ubuntu.
async fn start_elevator_server() {
    let ip_address = get_ip_address();
    let ssh_password = "Sanntid15"; // Hardkodet passord, vurder sikkerhetsrisiko

    if cfg!(target_os = "windows") {
        println!("Starter elevatorserver på Windows...");
        Command::new("cmd")
            .args(&["/C", "start", "elevatorserver"])
            .spawn()
            .expect("Failed to start elevator server");
    } else {
        println!("Starter elevatorserver på Linux...");

        let elevator_server_command = format!(
            "sshpass -p '{}' ssh student@{} 'nohup elevatorserver > /dev/null 2>&1 &'",
            ssh_password, ip_address
        );
        // Det starter serveren uten terminal. Om du vil avslutte serveren: pkill -f elevatorserver

        // Alternativt:                                    pgrep -f elevatorserver  # Finner PID (Process ID)
        //                                              kill <PID>            # Avslutter prosessen


        println!("\nStarter elevatorserver i ny terminal:\n\t{}", elevator_server_command);

        let _ = Command::new("sh")
            .arg("-c")
            .arg(&elevator_server_command)
            .output().await
            .expect("Feil ved start av elevatorserver");
    }

    println!("Elevator server startet.");
}

// ### Kjører den lokale heisen

/// Runs the local elevator
///
/// ## Parameters
/// `wv_watch_rx`: Rx on watch the worldview is being sent on in the system
/// `update_elev_state_tx`: mpsc sender used to update [local_network::update_wv_watch] when the elevator is in a new
state
```

```
/// `local_elev_tx`: mpsc sender used to update [local_network::update_wv_watch] when a message has been recieved form the elevator
///
/// ## Behavior
/// - The function starts the elevatorserver on the machine, and starts polling for messages
/// - The function starts a thread which forwards messages from the elevator to [local_network::update_wv_watch]
/// - The function starts a thread which executes the first task for your own elevator in the worldview
///
/// ## Note
/// This function loops over a tokio::yield_now(). This is added in case further implementation is added which makes the function permanently-blocking, forcing the user to spawn this function in a tokio task. In theroy, this could be removed, but for now: call this function asynchronously
pub async fn init(local_elev_tx: mpsc::Sender<elevio::ElevMessage>) -> e::Elevator {
    // Start elevator-serveren
    start_elevator_server().await;
    let local_elev_channels: LocalElevChannels = LocalElevChannels::new();
    let _ = sleep(config::SLAVE_TIMEOUT);
    let elevator: e::Elevator = e::Elevator::init(config::LOCAL_ELEV_IP, config::DEFAULT_NUM_FLOORS).expect("Feil!");

    // Start polling på meldinger fra heisen
    // ______START:: LESE KNAPPER_______________
    {
        let elevator = elevator.clone();
        tokio::spawn(async move {
            elevio::poll::call_buttons(elevator, local_elev_channels.txs.call_button, config::ELEV_POLL)
        });
    }
    {
        let elevator = elevator.clone();
        tokio::spawn(async move {
            elevio::poll::floor_sensor(elevator, local_elev_channels.txs.floor_sensor, config::ELEV_POLL)
        });
    }
    {
        let elevator = elevator.clone();
        tokio::spawn(async move {
            elevio::poll::stop_button(elevator, local_elev_channels.txs.obstruction, config::ELEV_POLL)
        });
    }
    {
        let elevator = elevator.clone();
        tokio::spawn(async move {
            elevio::poll::obstruction(elevator, local_elev_channels.txs.stop_button, config::ELEV_POLL)
        });
    }
    // ______STOPP:: LESE KNAPPER_______________

    {
        let _listen_task = tokio::spawn(async move {
            let _ = read_from_local_elevator(local_elev_channels.rxs, local_elev_tx).await;
        });
    }


    elevator
}

/// ### Videresender melding fra egen heis til update_wv
async fn read_from_local_elevator(rxs: LocalElevRxs, local_elev_tx: mpsc::Sender<elevio::ElevMessage>) -> std::io::Result<()> {
    loop {
        // Sjekker hver kanal med `try_recv()`
        if let Ok(call_button) = rxs.call_button.try_recv() {
            //println!("CB: {:#?}", call_button);
            let msg = elevio::ElevMessage {
                msg_type: elevio::ElevMsgType::CALLBTN,
                call_button: Some(call_button),
                floor_sensor: None,
                stop_button: None,
                obstruction: None,
            };
            let _ = local_elev_tx.send(msg).await;
        }

        if let Ok(floor) = rxs.floor_sensor.try_recv() {
            //println!("Floor: {:#?}", floor);
            let msg = elevio::ElevMessage {
                msg_type: elevio::ElevMsgType::FLOORSENS,
                call_button: None,
                floor_sensor: Some(floor),
                stop_button: None,
                obstruction: None,
            };
            let _ = local_elev_tx.send(msg).await;
        }

        if let Ok(stop) = rxs.stop_button.try_recv() {
            //println!("Stop button: {:#?}", stop);
            let msg = elevio::ElevMessage {
                msg_type: elevio::ElevMsgType::STOPBTN,
                call_button: None,
                floor_sensor: None,
                stop_button: Some(stop),
                obstruction: None,
            };
            let _ = local_elev_tx.send(msg).await;
        }

        if let Ok(obstr) = rxs.obstruction.try_recv() {
            //println!("Obstruction: {:#?}", obstr);
            let msg = elevio::ElevMessage {
                msg_type: elevio::ElevMsgType::OBSTRX,
                call_button: None,
                floor_sensor: None,
                stop_button: None,
                obstruction: Some(obstr),
            };
            let _ = local_elev_tx.send(msg).await;
        }

        // Kort pause for å unngå å spinne CPU unødvendig
        sleep(Duration::from_millis(10)).await;
    }
}



/// ### Handles messages from the local elevator
///
/// This function processes messages received from the local elevator and updates
/// the worldview accordingly. It supports different message types such as call
/// buttons, floor sensors, stop buttons, and obstruction notifications. It also
/// manages the state of the elevator container based on the received data.
///
/// ## Parameters
/// - `local_elev_rx`: A mutable reference to the mpsc reciever recieving messages sent from [read_from_local_elevator].
/// - `container`: A mutable reference to the elevatorcontainer.
///
/// ## Behavior
/// The function reads all available messages on the mpsc reciever. Then it performs different actions based on the type
of the message:
/// - **Call button (`CBTN`)**: Adds the call button to the `calls` list in the elevator container.
/// - **Floor sensor (`FSENS`)**: Updates the `last_floor_sensor` field in the elevator container.
/// - **Stop button (`SBTN`)**: A placeholder for future functionality to handle stop button messages.
/// - **Obstruction (`OBSTRX`)**: Sets the `obstruction` field in the elevator container to the
///   received value.
pub async fn update_elev_container_from_msgs(local_elev_rx: &mut mpsc::Receiver<elevio::ElevMessage>, container:
&mut ElevatorContainer, cab_call_timer: &mut Timer, error_timer: &mut Timer) {
    loop{
        match local_elev_rx.try_recv() {
            Ok(msg) => {
                match msg.msg_type {
                    elevio::ElevMsgType::CALLBTN => {
                        if let Some(call_btn) = msg.call_button {
                            print::info(format!("Callbutton: {:?}", call_btn));

                            match call_btn.call_type {
                                elevio::CallType::INSIDE => {
                                    cab_call_timer.release_timer();
                                    container.cab_requests[call_btn.floor as usize] = true;
                                }
                                elevio::CallType::UP => {
                                    container.unsent_hall_request[call_btn.floor as usize][0] = true;
                                }
                                elevio::CallType::DOWN => {
                                    container.unsent_hall_request[call_btn.floor as usize][1] = true;
                                }
                                elevio::CallType::COSMIC_ERROR => {},
                            }
                        }
                    }

                    elevio::ElevMsgType::FLOORSENS => {
                        print::info(format!("Floor: {:?}", msg.floor_sensor));
                        if let Some(floor) = msg.floor_sensor {
                            container.last_floor_sensor = floor;
                        }

                    }

                    elevio::ElevMsgType::STOPBTN => {
                        print::info(format!("Stop button: {:?}", msg.stop_button));
                        // TODO: selvforklarende
                    }

                    elevio::ElevMsgType::OBSTRX => {
                        print::info(format!("Obstruction: {:?}", msg.obstruction));
                        if let Some(obs) = msg.obstruction {
                            container.obstruction = obs;
                            if !obs && error_timer.timer_timeouted() {
                                error_timer.timer_start();
                                container.behaviour = ElevatorBehaviour::Idle; //må vekk
                            }
                        }
                    }
                }
            },
            Err(_) => {
                break;
            }
        }
    }

}
```

Innhald frå Rust-filer

Fil: elevator_pro_rebrand/src/elevator_logic/fsm.rs

```
use std::f32::consts::E;
use std::task;



use tokio::time::sleep;
use tokio::sync::mpsc;
use crate::{elevio::{self, elev::Elevator}, world_view, print};
use crate::world_view::{Dirn, ElevatorBehaviour, ElevatorContainer};
use crate::elevator_logic::self_elevator;
use crate::elevator_logic::request;
use crate::config;
use super::{lights, timer};


pub async fn onInit(self_container: &mut ElevatorContainer, e: Elevator, local_elev_rx: &mut mpsc::Receiver<elevio::ElevMessage>,
            cab_call_timer: &mut timer::Timer, error_timer: &mut timer::Timer, door_timer: &mut timer::Timer){

    e.motor_direction(Dirn::Down as u8);
    self_container.behaviour = ElevatorBehaviour::Moving;
    self_container.dirn = Dirn::Down;

    while self_container.last_floor_sensor == u8::MAX {
        self_elevator::update_elev_container_from_msgs( local_elev_rx, self_container, cab_call_timer , error_timer ).await;
        sleep(config::POLL_PERIOD).await;
    }
    onFloorArrival( self_container, e.clone(), door_timer, cab_call_timer).await;
}

pub async fn onFloorArrival(elevator: &mut ElevatorContainer, e: Elevator, door_timer: &mut timer::Timer, cab_call_timer: &mut timer::Timer) {
    // Ved init between floors: last_floor = 255, sett den til høyeste etasje for å slippe index error
    if elevator.last_floor_sensor > elevator.num_floors {
        elevator.last_floor_sensor = elevator.num_floors-1;
    }

    lights::set_cab_light(e.clone(), elevator.last_floor_sensor);

    match elevator.behaviour {
        ElevatorBehaviour::Moving | ElevatorBehaviour::Error => {
            if request::should_stop(&elevator.clone()) {
                e.motor_direction(Dirn::Stop as u8);
                println!("floor: {}", elevator.last_floor_sensor);
                request::clear_at_current_floor(elevator);
                lights::set_door_open_light(e);
                door_timer.timer_start();
                cab_call_timer.timer_start();
                elevator.behaviour = ElevatorBehaviour::DoorOpen;
            }
        }
        _ => {},
    }
}

pub async fn onDoorTimeout(elevator: &mut ElevatorContainer, e: Elevator, cab_call_timer: &mut timer::Timer) {
    match elevator.behaviour {
        ElevatorBehaviour::DoorOpen => {
            let DBPair = request::choose_direction(&elevator.clone());


            elevator.behaviour = DBPair.behaviour;
            elevator.dirn = DBPair.dirn;

            match elevator.behaviour {
                ElevatorBehaviour::DoorOpen => {
                    request::clear_at_current_floor(elevator);
                }
                _ => {
                    lights::clear_door_open_light(e.clone());
                    e.motor_direction(elevator.dirn as u8);
                }
            }
        },
        _ => {},
    }
}
```

Innhald frå Rust-filer

Fil: elevator_pro_rebrand/src/elevator_logic/mod.rs

```
pub mod fsm;
pub mod request;
pub mod timer;
mod lights;
mod self_elevator;

use std::time::Duration;

use tokio::task::yield_now;
use tokio::sync::mpsc;
use tokio::sync::watch;
use tokio::time::sleep;
use crate::config;
use crate::elevio;
use crate::elevio::elev::Elevator;
use crate::elevio::ElevMessage;
use crate::print;
use crate::world_view;
use crate::world_view::Dirn;
use crate::world_view::ElevatorBehaviour;
use crate::world_view::ElevatorContainer;

/// Initializes and runs the local elevator logic as a set of async tasks.
///
/// This function performs the following:
/// - Initializes the local elevator instance and communication channels.
/// - Spawns one async task to handle elevator state and behavior (`handle_elevator`).
/// - Spawns another task to continuously update the hall request lights based on world view state.
/// - Keeps the main task alive indefinitely via an infinite `yield_now` loop.
///
/// # Parameters
/// - `wv_watch_rx`: A `watch::Receiver` that provides the latest serialized world view.
/// - `elevator_states_tx`: A `mpsc::Sender` used to send the local elevator state back to the system.
///
/// # Behavior
/// - Runs all logic asynchronously and non-blocking.
/// - Continues operation until externally cancelled or interrupted.
/// - Each spawned task operates independently of the main loop.
///
/// # Note
/// The hall light updater task continuously reads the world view and sets the hall lights based on
/// the current state of the local elevator. Failure to extract the local container results in a warning.
pub async fn run_local_elevator(wv_watch_rx: watch::Receiver<Vec<u8>>, elevator_states_tx: mpsc::Sender<Vec<u8>>) {
    let (local_elev_tx, local_elev_rx) = mpsc::channel::<ElevMessage>(100);

    let elevator = self_elevator::init(local_elev_tx).await;



    // Task som utfører deligerte tasks (ikke implementert korrekt enda)
    {
        let elevator_c = elevator.clone();
        let wv_watch_rx_c = wv_watch_rx.clone();
        let _handle_task = tokio::spawn(async move {
            let _ = handle_elevator(wv_watch_rx_c, elevator_states_tx, local_elev_rx, elevator_c).await;
        });
    }

    {
        let e = elevator.clone();
        let wv_watch_rx_c = wv_watch_rx.clone();
        // Task som setter på hall_lights
        tokio::spawn(async move {
            let mut wv = world_view::get_wv(wv_watch_rx);
            loop {
                world_view::update_wv(wv_watch_rx_c.clone(), &mut wv).await;
                match world_view::extract_self_elevator_container(wv.clone()) {
                    Some(self_elevator) => {
                        lights::set_hall_lights(wv.clone(), e.clone(), &self_elevator);
                    }
                    None => {
                        print::warn(format!("Failed to extract self elevator container"));
                    }
                }
                sleep(config::POLL_PERIOD).await;
            }
        });
    }

    loop {
        yield_now().await;
    }

}


/// Main event loop for handling local elevator logic, state transitions, and communication.
///
/// This function implements the core elevator state machine and handles:
/// - Receiving updates from local hardware (buttons, floor sensors, etc.)
/// - Executing FSM transitions based on current state and events
/// - Managing timers for door state, cab call delays, and error detection
/// - Updating direction and motor control
/// - Sending updated elevator state to the rest of the system
/// - Applying updates from the world view (task assignments, shared state)
///
/// # Parameters
/// - `wv_watch_rx`: A `watch::Receiver` used to access the latest global world view.
/// - `elevator_states_tx`: A `mpsc::Sender` used to transmit updated local elevator state.
/// - `local_elev_rx`: A `mpsc::Receiver` that receives elevator hardware messages.
/// - `e`: Handle representing the elevator hardware interface (for lights, motor, etc.)
///
```

```
/// # Behavior
/// - Blocks in a loop, continuously reacting to inputs and updating state.
/// - Relies on helper functions for modular FSM logic and safety mechanisms.
/// - Polls the world view and local state at a fixed interval (`config::POLL_PERIOD`).
///
/// # Notes
/// - The function will attempt to initialize the elevator state by waiting for it
///   to reach the closest floor in downward direction (via `fsm::onInit`).
/// - If the elevator starts on floor 0, special care must be taken (known crash case).
/// - Errors are handled internally via timers and behavior transitions.
pub async fn handle_elevator(wv_watch_rx: watch::Receiver<Vec<u8>>, elevator_states_tx: mpsc::Sender<Vec<u8>>,
mut local_elev_rx: mpsc::Receiver<elevio::ElevMessage>, e: Elevator) {

    let mut wv = world_view::get_wv(wv_watch_rx.clone());
    let mut self_container = await_valid_self_container(wv_watch_rx.clone()).await;



    let mut door_timer = timer::new(Duration::from_secs(3));
    let mut cab_call_timer = timer::new(Duration::from_secs(10));
    let mut error_timer = timer::new(Duration::from_secs(7));
    let mut prev_cab_call_timer_stat:bool = false;

    //init the state. this is blocking until we reach the closest foor in down direction
        fsm::onInit(&mut self_container, e.clone(), &mut local_elev_rx, &mut cab_call_timer, &mut error_timer, &mut
door_timer).await;

    // self_container.dirn = Dirn::Stop;
    let mut prev_behavior:ElevatorBehaviour = self_container.behaviour;
    let mut prev_floor: u8 = self_container.last_floor_sensor;

    loop {
        /*OBS OBS!! krasjer når vi starter i 0 etasje..... uff da */
        //Les nye data fra heisen, putt de inn i self_container

         self_elevator::update_elev_container_from_msgs(&mut local_elev_rx, &mut self_container, &mut cab_call_timer ,
&mut error_timer ).await;

        /*==================================================================*/
        /*                        START: FSM Events                         */
        /*==================================================================*/
        handle_floor_sensor_update(
            &mut self_container,
            e.clone(),
            &mut prev_floor,
            &mut door_timer,
            &mut cab_call_timer,
            &mut error_timer,
        ).await;


        handle_door_timeout_and_lights(
            &mut self_container,
            e.clone(),
            &door_timer,
            &mut cab_call_timer,
        ).await;

        handle_error_timeout(
            &self_container,
            &cab_call_timer,
            &mut error_timer,
            prev_cab_call_timer_stat,
        );

        // fsm::onIdle ?
        handle_idle_state(&mut self_container, e.clone(), &mut door_timer);
        /*======================================================================*/
        /*                         END: FSM Events                         */
        /*======================================================================*/


/*==========================================================================================
==================================================*/

        update_motor_direction_if_needed(&self_container, &e);

        update_error_state(&mut self_container, &error_timer, &mut prev_cab_call_timer_stat);

        let last_behavior: ElevatorBehaviour = track_behavior_change(&self_container, &mut prev_behavior);
        stop_motor_on_dooropen_to_error(&mut self_container, last_behavior, prev_behavior);



        //Send til update_wv -> nye self_container
        let _ = elevator_states_tx.send(world_view::serial::serialize_elev_container(&self_container)).await;

        //Hent nyeste worldview
        if world_view::update_wv(wv_watch_rx.clone(), &mut wv).await{
            update_tasks_and_hall_requests(&mut self_container, wv.clone()).await;
        }
        yield_now().await;
        sleep(config::POLL_PERIOD).await;



    }
}

/// Updates the local elevator container's task-related fields based on the latest world view.
///
/// This function attempts to extract the elevator container corresponding to `SELF_ID` from
/// the given serialized world view. If found, it updates `tasks` and `unsent_hall_request`
/// in the local container. If extraction fails, the local values are left unchanged,
/// and a warning is printed.
```

```
///
/// # Parameters
/// - `self_container`: A mutable reference to the local elevator container to be updated.
/// - `wv`: A serialized world view (`Vec<u8>`) to extract the container from.
///
/// # Behavior
/// - Safe to call repeatedly.
/// - Only updates the two mentioned fields if a valid container is found.
/// - Prints a warning if no container is found.
///
/// # Example
/// ```ignore
/// update_tasks_and_hall_requests(&mut local_container, serialized_worldview).await;
/// ```
async fn update_tasks_and_hall_requests(self_container: &mut ElevatorContainer, wv: Vec<u8>){
    if let Some(task_container) = world_view::extract_self_elevator_container(wv) {
        self_container.tasks = task_container.tasks;
        self_container.unsent_hall_request = task_container.unsent_hall_request;
    } else {
        print::warn(format!("Failed to extract self elevator container  keeping previous value"));
    }
}

/// Continuously attempts to extract the local elevator container from the world view until successful.
///
/// This function loops until it successfully extracts the container for `SELF_ID` from the
/// current world view received over a `watch::Receiver`. It prints a warning for each failed
/// attempt and waits 100 milliseconds between retries.
///
/// # Parameters
/// - `wv_rx`: A watch channel receiver providing the latest serialized world view (`Vec<u8>`).
///
/// # Returns
/// - A fully initialized `ElevatorContainer` once it is successfully extracted.
///
/// # Notes
/// - This function does not return until a valid container is available.
/// - It is suitable for running inside a long-lived async task.
///
/// # Example
/// ```ignore
/// let container = await_valid_self_container(wv_rx).await;
/// ```
async fn await_valid_self_container(wv_rx: watch::Receiver<Vec<u8>>) -> ElevatorContainer {
    loop {
        let wv = world_view::get_wv(wv_rx.clone());
        if let Some(container) = world_view::extract_self_elevator_container(wv) {
            return container;
        } else {
            print::warn(format!("Failed to extract self elevator container, retrying..."));
            sleep(Duration::from_millis(100)).await;
        }
```

```
    }
}


// Hjelpefunksjona til loopen
/// Handles floor arrival updates when a new floor sensor reading is detected.
///
/// If the current floor (`last_floor_sensor`) is different from the previous floor,
/// this function calls the floor arrival state handler and starts the error timer.
/// If the request came from an inside button, the cab call timer is also released.
///
/// # Parameters
/// - `self_container`: The mutable elevator container representing the local elevator.
/// - `e`: Elevator identifier or handle.
/// - `prev_floor`: Mutable reference to the previously seen floor.
/// - `door_timer`: Timer used to control door timing.
/// - `cab_call_timer`: Timer tracking inside requests.
/// - `error_timer`: Timer for detecting inactivity or errors.
///
/// # Behavior
/// - Updates the previous floor value.
/// - Triggers door and cab call logic based on sensor input.
pub async fn handle_floor_sensor_update(
    self_container: &mut ElevatorContainer,
    e: Elevator,
    prev_floor: &mut u8,
    door_timer: &mut timer::Timer,
    cab_call_timer: &mut timer::Timer,
    error_timer: &mut timer::Timer,
) {
    if *prev_floor != self_container.last_floor_sensor {
        fsm::onFloorArrival(self_container, e, door_timer, cab_call_timer).await;
        error_timer.timer_start();

        // Skal ignorere cab_call_timer viss oppdraget kom frå ein inside-knapp
        if !request::was_outside(self_container) {
            cab_call_timer.release_timer();
        }
        *prev_floor = self_container.last_floor_sensor;
    }
}

/// Handles door timeout logic and clears the door light when appropriate.
///
/// If the door timer has expired and no obstruction is detected, this function clears
/// the door open light. If the elevator is moving toward a cab call, the cab call timer
/// is released. If the cab call timer has also expired, the system proceeds to handle
/// the door timeout state transition.
///
/// # Parameters
/// - `self_container`: Mutable reference to the elevator's internal state.
/// - `e`: Elevator identifier or hardware handle used to control lights and motors.
/// - `door_timer`: Timer that tracks how long the door has been open.
/// - `cab_call_timer`: Timer used to delay door close for cab calls.
///
/// # Behavior
/// - Clears door light after timeout.
/// - Handles door-close logic via finite state machine if cab call timer is also expired.
async fn handle_door_timeout_and_lights(
    self_container: &mut ElevatorContainer,
    e: Elevator,
    door_timer: &timer::Timer,
    cab_call_timer: &mut timer::Timer,
) {
    if door_timer.timer_timeouted() && !self_container.obstruction {
        lights::clear_door_open_light(e.clone());

        if request::moving_towards_cab_call(&self_container.clone()) {
            cab_call_timer.release_timer();
        }

        if cab_call_timer.timer_timeouted() {
            fsm::onDoorTimeout(self_container, e.clone(), cab_call_timer).await;
        }
    }
}

/// Monitors elevator activity and triggers error behavior after a timeout period.
///
/// If no cab call has timed out or the elevator is idle, the error timer is restarted.
/// If the error timer itself has expired and a cab call was previously active,
/// the elevator enters an error state and logs a critical error message.
///
/// # Parameters
/// - `self_container`: Reference to the elevator state being monitored.
/// - `cab_call_timer`: Timer tracking how long a cab call has been pending.
/// - `error_timer`: Mutable timer for detecting inactivity or system faults.
/// - `prev_cab_call_timer_stat`: Whether the cab call timer had previously expired.
///
/// # Behavior
/// - Triggers an error state if prolonged inactivity or failure is detected.
fn handle_error_timeout(
    self_container: &ElevatorContainer,
    cab_call_timer: &timer::Timer,
    error_timer: &mut timer::Timer,
    prev_cab_call_timer_stat: bool,
) {
    if !cab_call_timer.timer_timeouted() || self_container.behaviour == ElevatorBehaviour::Idle {
        error_timer.timer_start();
    }

    if error_timer.timer_timeouted() && !prev_cab_call_timer_stat {
        print::cosmic_err("Feil på travel!!!!".to_string());
    }
```

```
}

/// Attempts to transition the elevator from idle to active movement if a request is pending.
///
/// If the elevator is currently idle, the system chooses a new direction and behavior
/// using the request logic. If a non-idle state is chosen, the elevator's direction
/// and behavior are updated, the door timer is started, and the motor is stopped
/// in preparation for movement or door logic.
///
/// # Parameters
/// - `self_container`: Mutable reference to the elevator's current state.
/// - `e`: Elevator handle or control interface.
/// - `door_timer`: Timer used to delay transitions or prepare door actions.
///
/// # Behavior
/// - Only operates when the elevator is in an idle state.
/// - Initializes direction and behavior when transitioning out of idle.
/// - Starts door timer and stops the motor to stabilize before further action.
fn handle_idle_state(
    self_container: &mut ElevatorContainer,
    e: Elevator,
    door_timer: &mut timer::Timer,
) {
    if self_container.behaviour == ElevatorBehaviour::Idle {
        let DBPair = request::choose_direction(&self_container.clone());

        if DBPair.behaviour != ElevatorBehaviour::Idle {
            print::err(format!("Skal nå være: {:?}", DBPair.behaviour));
            self_container.dirn = DBPair.dirn;
            self_container.behaviour = DBPair.behaviour;
            door_timer.timer_start();
            e.motor_direction(Dirn::Stop as u8);
        }
    }
}

/// Updates the motor direction if the elevator is not in the DoorOpen state.
///
/// This function sends the current direction (`dirn`) to the motor controller
/// only if the elevator is not in the `DoorOpen` state.
///
/// # Parameters
/// - `self_container`: Reference to the current elevator state.
/// - `e`: Elevator interface used to send motor direction commands.
///
/// # Behavior
/// - Prevents motor updates while the door is open.
/// - Useful for ensuring motor is only active during appropriate states.
pub fn update_motor_direction_if_needed(self_container: &ElevatorContainer, e: &Elevator) {
    if self_container.behaviour != ElevatorBehaviour::DoorOpen {
        e.motor_direction(self_container.dirn as u8);
    }
```

```rust
}

/// Updates the elevator state based on the error timer's status.
///
/// If the error timer has expired, the elevator transitions into the `Error` state
/// and the `prev_cab_call_timer_stat` flag is set. Otherwise, the flag is cleared.
///
/// # Parameters
/// - `self_container`: Mutable reference to the elevator state.
/// - `error_timer`: Timer that tracks potential error conditions.
/// - `prev_cab_call_timer_stat`: Mutable flag to track whether the system was previously in a faultable state.
///
/// # Behavior
/// - Sets elevator to `Error` if timer has expired.
/// - Updates a boolean tracking previous timer state.
pub fn update_error_state(
    self_container: &mut ElevatorContainer,
    error_timer: &timer::Timer,
    prev_cab_call_timer_stat: &mut bool,
) {
    if error_timer.timer_timeouted() {
        *prev_cab_call_timer_stat = true;
        self_container.behaviour = ElevatorBehaviour::Error;
    } else {
        *prev_cab_call_timer_stat = false;
    }
}

/// Tracks and logs changes to the elevator's behavior state.
///
/// Compares the current elevator behavior to a previously stored value.
/// If the state has changed, logs the transition and updates `prev_behavior`.
///
/// # Parameters
/// - `self_container`: Reference to the current elevator state.
/// - `prev_behavior`: Mutable reference to the last recorded behavior state.
///
/// # Returns
/// - The previous behavior before the update (if any).
///
/// # Behavior
/// - Detects and logs behavior transitions for debugging or system monitoring.
pub fn track_behavior_change(
    self_container: &ElevatorContainer,
    prev_behavior: &mut ElevatorBehaviour,
) -> ElevatorBehaviour {
    let last_behavior = *prev_behavior;

    if *prev_behavior != self_container.behaviour {
        *prev_behavior = self_container.behaviour;
        println!("Endra status: {:?} -> {:?}", last_behavior, self_container.behaviour);
    }
    last_behavior
}

/// Forces the elevator to stop the motor when transitioning from DoorOpen to Error state.
///
/// If the behavior transition is specifically from `DoorOpen` to `Error`, the elevator
/// direction is set to `Stop` to ensure the motor halts immediately.
///
/// # Parameters
/// - `self_container`: Mutable reference to the elevator state.
/// - `last_behavior`: The previous elevator behavior before the transition.
/// - `current_behavior`: The current elevator behavior after the transition.
///
/// # Behavior
/// - Stops the motor only for the specific transition from `DoorOpen`  `Error`.
pub fn stop_motor_on_dooropen_to_error(
    self_container: &mut ElevatorContainer,
    last_behavior: ElevatorBehaviour,
    current_behavior: ElevatorBehaviour,
) {
    if last_behavior == ElevatorBehaviour::DoorOpen && current_behavior == ElevatorBehaviour::Error {
        self_container.dirn = Dirn::Stop;
    }
}
```

Innhald frå Rust-filer

Fil: elevator_pro_rebrand/src/elevator_logic/timer.rs

```
pub struct Timer {
    hard_timeout: bool,
    timer_active: bool,
    timeout_duration: tokio::time::Duration,
    start_time: tokio::time::Instant,
}

pub fn new(timeout_duration: tokio::time::Duration) -> Timer {
    Timer{
        hard_timeout: false,
        timer_active: false,
        timeout_duration: timeout_duration,
        start_time: tokio::time::Instant::now(),
    }
}
impl Timer {
    pub fn timer_start(&mut self) {
        self.hard_timeout = false;
        self.timer_active = true;
        self.start_time = tokio::time::Instant::now();
    }

    pub fn release_timer(&mut self) {
        self.hard_timeout = true;
    }

    pub fn get_wall_time(&mut self) -> tokio::time::Duration {
        return tokio::time::Instant::now() - self.start_time
    }

    pub fn timer_timeouted(&self) -> bool {
        return (self.timer_active && (tokio::time::Instant::now() - self.start_time) > self.timeout_duration) || self.hard_timeout;
    }
}
```

Innhald frå Rust-filer

Fil: elevator_pro_rebrand/src/world_view/serial.rs

```
//! Serialization and Deserialization for [WorldView] and [ElevatorContainer]

use crate::world_view::{WorldView, ElevatorContainer};
use crate::print;


/// Serializes a `WorldView` into a binary format.
///
/// Uses `bincode` for efficient serialization.
/// If serialization fails, the function logs the error and panics.
///
/// ## Parameters
/// - `worldview`: A reference to the `WorldView` to be serialized.
///
/// ## Returns
/// - A `Vec<u8>` containing the serialized data.
pub fn serialize_worldview(worldview: &WorldView) -> Vec<u8> {
    let encoded = bincode::serialize(worldview);
    match encoded {
        Ok(serialized_data) => {
            // Deserialisere WorldView fra binært format
            return serialized_data;
        }
        Err(e) => {
            println!("{:?}", worldview);
            print::err(format!("Serialization failed: {} (world_view.rs, serialize_worldview())", e));
            panic!();
        }
    }
}

/// Deserializes a `WorldView` from a binary format.
///
/// Uses `bincode` for deserialization.
/// If deserialization fails, the function logs the error and panics.
///
/// ## Parameters
/// - `data`: A byte slice (`&[u8]`) containing the serialized `WorldView`.
///
/// ## Returns
/// - A `WorldView` instance reconstructed from the binary data.
pub fn deserialize_worldview(data: &[u8]) -> WorldView {
    let decoded = bincode::deserialize(data);


    match decoded {
        Ok(serialized_data) => {
            // Deserialisere WorldView fra binært format
            return serialized_data;
        }
        Err(e) => {
            print::err(format!("Serialization failed: {} (world_view.rs, deserialize_worldview())", e));
            panic!();
        }
    }
}

/// Serializes an `ElevatorContainer` into a binary format.
///
/// Uses `bincode` for serialization.
/// If serialization fails, the function logs the error and panics.
///
/// ## Parameters
/// - `elev_container`: A reference to the `ElevatorContainer` to be serialized.
///
/// ## Returns
/// - A `Vec<u8>` containing the serialized data.
pub fn serialize_elev_container(elev_container: &ElevatorContainer) -> Vec<u8> {
    let encoded = bincode::serialize(elev_container);
    match encoded {
        Ok(serialized_data) => {
            // Deserialisere WorldView fra binært format
            return serialized_data;
        }
        Err(e) => {
            print::err(format!("Serialization failed: {} (world_view.rs, serialize_elev_container())", e));
            panic!();
        }
    }
}

/// Deserializes an `ElevatorContainer` from a binary format.
///
/// Uses `bincode` for deserialization.
/// If deserialization fails, the function logs the error and panics.
///
/// ## Parameters
/// - `data`: A byte slice (`&[u8]`) containing the serialized `ElevatorContainer`.
///
/// ## Returns
/// - An `ElevatorContainer` instance reconstructed from the binary data.
pub fn deserialize_elev_container(data: &[u8]) -> ElevatorContainer {
    let decoded = bincode::deserialize(data);


    match decoded {
        Ok(serialized_data) => {
            // Deserialisere WorldView fra binært format
            return serialized_data;
        }
        Err(e) => {
            print::err(format!("Serialization failed: {} (world_view.rs, deserialize_elev_container())", e));
            panic!();
        }
    }
}
```

Innhald frå Rust-filer

Fil: elevator_pro_rebrand/src/world_view/mod.rs

```rust
pub mod serial;

use serde::{Serialize, Deserialize};
use tokio::sync::watch;
use std::collections::HashMap;
use std::sync::atomic::Ordering;
use crate::config;
use crate::print;
use crate::network;



#[allow(missing_docs)]
#[derive(Debug, Clone, Copy, PartialEq, Eq, Serialize, Deserialize)]
/// Struct describing direction an elevator is taking calls in.
pub enum Dirn {
    Down = -1,
    Stop = 0,
    Up = 1,
}

#[allow(missing_docs)]
#[derive(Debug, Clone, Copy, PartialEq, Eq, Serialize, Deserialize)]
/// Struct describing the current behaviour of an elevator
pub enum ElevatorBehaviour {
    Idle,
    Moving,
    DoorOpen,
    Error,
}


/// Represents the state of an elevator, including tasks, status indicators, and movement.
#[derive(Serialize, Deserialize, Debug, Clone)]
pub struct ElevatorContainer {
    /// Unique identifier for the elevator.
    /// Default: [config::ERROR_ID]
    pub elevator_id: u8,

    /// The number of floors the elevator can access
    /// Default: [config::DEFAULT_NUM_FLOORS]
    pub num_floors: u8,

    /// Vector of hall requests not yet sent to master over TCP
    /// Default: full of \[false, false\], length [config::DEFAULT_NUM_FLOORS]
    pub unsent_hall_request: Vec<[bool; 2]>,

    /// Vector of cab_requests.
    /// Default: full of false, length [config::DEFAULT_NUM_FLOORS]
    pub cab_requests: Vec<bool>,
    /// Vector of hall_requests given to this elevator from the manager.
    /// Default: full of \[false, false\], length [config::DEFAULT_NUM_FLOORS]
    pub tasks: Vec<[bool; 2]>,

    /// [Dirn]
    ///  Default: [Dirn::Stop]
    pub dirn: Dirn,

    /// The current behaviour of the elevator
    /// Default: [ElevatorBehaviour::Idle]
    pub behaviour: ElevatorBehaviour,

    /// Indicates whether the elevator detects an obstruction.
    /// Default: false
    pub obstruction: bool,

    /// The last detected floor sensor position.
    /// Default: 255
    pub last_floor_sensor: u8,
}


impl Default for ElevatorContainer {
    fn default() -> Self {
        Self {
            elevator_id: config::ERROR_ID,
            num_floors: config::DEFAULT_NUM_FLOORS,
            unsent_hall_request: vec![[false; 2]; config::DEFAULT_NUM_FLOORS as usize],
            cab_requests: vec![false; config::DEFAULT_NUM_FLOORS as usize],
            tasks: vec![[false, false]; config::DEFAULT_NUM_FLOORS as usize],
            dirn: Dirn::Stop,
            behaviour: ElevatorBehaviour::Idle,
            obstruction: false,
            last_floor_sensor: 255,
        }
    }
}


/// Represents the system's current state (WorldView).
///
/// `WorldView` contains an overview of all elevators in the system,
/// the master elevator's ID, and the call buttons pressed outside the elevators.
#[derive(Serialize, Deserialize, Debug, Clone)]
pub struct WorldView {
    /// Number of elevators in the system.
    n: u8,
    /// The ID of the master elevator.
    pub master_id: u8,
    /// A vector contining statuses on all hall requests
    pub hall_request: Vec<[bool; 2]>,
    /// A list of `ElevatorContainer` structures containing
    ///   individual elevator information.
    pub elevator_containers: Vec<ElevatorContainer>,

    /// A HashMap backing up cab_call statuses for all elevators, mapping them to their IDs
    pub cab_requests_backup: HashMap<u8, Vec<bool>>,
}


impl Default for WorldView {
    /// Creates a default `WorldView` instance with no elevators and an invalid master ID.
    fn default() -> Self {
        Self {
            n: 0,
            master_id: config::ERROR_ID,
            // pending_tasks: Vec::new(),
            hall_request: vec![[false; 2]; config::DEFAULT_NUM_FLOORS as usize],
            elevator_containers: Vec::new(),
            cab_requests_backup: HashMap::new(),
        }
    }
}


impl WorldView {
    /// Adds an elevator to the system.
    ///
    /// Updates the number of elevators (`n`) accordingly.
    ///
    /// ## Parameters
    /// - `elevator`: The `ElevatorContainer` to be added.
    pub fn add_elev(&mut self, elevator: ElevatorContainer) {
        self.elevator_containers.push(elevator);
        self.n = self.elevator_containers.len() as u8;
    }

    /// Removes an elevator with the given ID from the system.
    ///
    /// If no elevator with the specified ID is found, a warning is printed.
    ///
    /// ## Parameters
    /// - `id`: The ID of the elevator to remove.
    pub fn remove_elev(&mut self, id: u8) {
        let initial_len = self.elevator_containers.len();

        self.elevator_containers.retain(|elevator| elevator.elevator_id != id);

        if self.elevator_containers.len() == initial_len {
            print::warn(format!("No elevator with ID {} was found. (remove_elev())", id));
        } else {
            print::ok(format!("Elevator with ID {} was removed. (remove_elev())", id));
        }
        self.n = self.elevator_containers.len() as u8;
    }

    /// Returns the number of elevators in the system.
    pub fn get_num_elev(&self) -> u8 {
        return self.n;
    }


    /// Sets the number of elevators manually.
    ///
    /// **Note:** This does not affect the `elevator_containers` list.
    /// Use `add_elev()` or `remove_elev()` to modify the actual elevators.
    ///
    /// ## Parameters
    /// - `n`: The new number of elevators.
    // TODO: Burde være veldig mulig å gjøre denne privat
    pub fn set_num_elev(&mut self, n: u8)  {
        self.n = n;
    }
}





/// Retrieves the index of an `ElevatorContainer` with the specified `id` in the deserialized `WorldView`.
///
/// This function deserializes the provided `WorldView` data and iterates through the elevator containers
/// to find the one that matches the given `id`. If found, it returns the index of the container; otherwise, it returns `None`.
///
/// ## Parameters
/// - `id`: The ID of the elevator whose index is to be retrieved.
/// - `wv`: A serialized `WorldView` as a `Vec<u8>`.
///
/// ## Returns
/// - `Some(usize)`: The index of the `ElevatorContainer` in the `WorldView` if found.
/// - `None`: If no elevator with the given `id` exists.
pub fn get_index_to_container(id: u8, wv: Vec<u8>) -> Option<usize> {
    let wv_deser = serial::deserialize_worldview(&wv);
    for i in 0..wv_deser.get_num_elev() {
        if wv_deser.elevator_containers[i as usize].elevator_id == id {
            return Some(i as usize);
        }
    }
    return None;
}


/// Fetches a clone of the latest local worldview (wv) from the system.
///
/// This function retrieves the most recent worldview stored in the provided `LocalChannels` object.
/// It returns a cloned vector of bytes representing the current serialized worldview.
///
/// # Parameters
/// - `chs`: The `LocalChannels` object, which contains the latest worldview data in `wv`.
///
/// # Return Value
/// Returns a vector of `u8` containing the cloned serialized worldview.
///
/// # Example
/// ```
/// use elevatorpro::utils::get_wv;
/// use elevatorpro::network::local_network::LocalChannels;
///
/// let local_chs = LocalChannels::new();
/// let _ = local_chs.watches.txs.wv.send(vec![1, 2, 3, 4]);
///
/// let fetched_wv = get_wv(local_chs.clone());
/// assert_eq!(fetched_wv, vec![1, 2, 3, 4]);
/// ```
///
/// **Note:** This function clones the current state of `wv`, so any future changes to `wv` will not affect the returned vector.
pub fn get_wv(wv_watch_rx: watch::Receiver<Vec<u8>>) -> Vec<u8> {
    wv_watch_rx.borrow().clone()
}

/// Asynchronously updates the worldview (wv) in the system.
///
/// This function reads the latest worldview data from a specific channel and updates
/// the given `wv` vector with the new data if it has changed. The function operates asynchronously,
/// allowing it to run concurrently with other tasks without blocking.
///
/// ## Parameters
/// - `chs`: The `LocalChannels` object, which holds the channels used for receiving worldview data.
/// - `wv`: A mutable reference to the `Vec<u8>` that will be updated with the latest worldview data.
///
/// ## Returns
/// - `true` if wv was updated, `false` otherwise.
///
/// ## Example
/// ```
/// # use tokio::runtime::Runtime;
/// use elevatorpro::utils::update_wv;
/// use elevatorpro::network::local_network::LocalChannels;
///
/// let chs = LocalChannels::new();
/// let mut wv = vec![1, 2, 3, 4];
///
/// # let rt = Runtime::new().unwrap();
/// # rt.block_on(async {///
/// chs.watches.txs.wv.send(vec![4, 3, 2, 1]);
/// let result = update_wv(chs.clone(), &mut wv).await;
```

```
/// assert_eq!(result, true);
/// assert_eq!(wv, vec![4, 3, 2, 1]);
///
/// let result = update_wv(chs.clone(), &mut wv).await;
/// assert_eq!(result, false);
/// assert_eq!(wv, vec![4, 3, 2, 1]);
/// # });
/// ```
///
/// ## Notes
/// - This function is asynchronous and requires an async runtime, such as Tokio, to execute.
/// - The `LocalChannels` channels allow for thread-safe communication across threads.
pub async fn update_wv(wv_watch_rx: watch::Receiver<Vec<u8>>, wv: &mut Vec<u8>) -> bool {
    let new_wv = wv_watch_rx.borrow().clone();  // Clone the latest data
    if new_wv != *wv {  // Check if the data has changed compared to the current state
        *wv = new_wv;  // Update the worldview if it has changed
        return true;
    }
    false
}


/// Checks if the current system is the master based on the latest worldview data.
///
/// This function compares the system's `SELF_ID` with the value at `MASTER_IDX` in the provided worldview (`wv`).
///
/// ## Returns
/// - `true` if the current system's `SELF_ID` matches the value at `MASTER_IDX` in the worldview.
/// - `false` otherwise.
pub fn is_master(wv: Vec<u8>) -> bool {
    return network::read_self_id() == wv[config::MASTER_IDX];
}

// /// Retrieves the latest elevator tasks from the system.
// ///
// /// This function borrows the value from the `elev_task` channel and clones it, returning a copy of the tasks.
// /// It is used to fetch the current tasks for the local elevator.
// ///
// /// ## Parameters
// /// - `chs`: A `LocalChannels` struct that contains the communication channels for the system.
// ///
// /// ## Returns
// /// - A `Vec<Task>` containing the current elevator tasks.
// pub fn get_elev_tasks(elev_task_rx: watch::Receiver<Vec<Task>>) -> Vec<Task> {
//     elev_task_rx.borrow().clone()
// }


/// Extracts the elevator container with the specified `id` from the given serialized worldview.
///
/// This function deserializes the provided worldview, filters out elevator containers
/// that do not match the given `id`, and returns the first matching result if available.
```

```
///
/// ## Parameters
/// - `wv`: A `Vec<u8>` representing the serialized worldview.
/// - `id`: The elevator ID to search for.
///
/// ## Returns
/// - `Some(ElevatorContainer)` if a container with the given `id` is found.
/// - `None` if no matching elevator container exists in the worldview.
///
/// ## Note
/// If multiple containers have the same `id`, only the first match is returned.
pub fn extract_elevator_container(wv: Vec<u8>, id: u8) -> Option<ElevatorContainer> {
    let mut deser_wv = serial::deserialize_worldview(&wv);

    deser_wv.elevator_containers.retain(|elevator| elevator.elevator_id == id);
    deser_wv.elevator_containers.get(0).cloned()
}

/// Retrieves a clone of the `ElevatorContainer` with `SELF_ID` from the latest worldview.
///
/// This function calls `extract_elevator_container` with `SELF_ID` to fetch the elevator container that matches the
/// current `SELF_ID` from the provided worldview (`wv`). The `SELF_ID` is a static identifier loaded from memory,
/// which represents the current elevator's unique identifier.
///
/// ## Parameters
/// - `wv`: The latest worldview in serialized state.
///
/// ## Returns
/// - A clone of the `ElevatorContainer` associated with `SELF_ID`.
///
/// **Note:** This function internally calls `extract_elevator_container` to retrieve the correct elevator container.
pub fn extract_self_elevator_container(wv: Vec<u8>) -> Option<ElevatorContainer> {
    let id = network::read_self_id();
    extract_elevator_container(wv, id)
}
```

Innhald frå Rust-filer

Fil: elevator_pro_rebrand/src/manager/json_serial.rs

```
// Library that allows us to use environment variables or command-line arguments to pass variables from terminal to the
program directly
use std::{collections::HashMap, env};
use serde::{Serialize, Deserialize};
use std::fs::File;
use std::io::Write;
// Library for executing terminal commands
use tokio::process::Command;
use crate::world_view::{self, ElevatorBehaviour};

#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct ElevatorState {
    behaviour: String,
    floor: i32,
    direction: String,
    cabRequests: Vec<bool>,
}
#[derive(Serialize, Deserialize)]
struct HallRequests {
    hallRequests: Vec<[bool; 2]>,
    states: HashMap<String, ElevatorState>,
}
// Function to execute the algorithm

pub async fn run_cost_algorithm(json_str: String) -> String {
    let cost_path = env::current_dir()
        .unwrap()
        .join("libs")
        .join("Project_resources")
        .join("cost_fns")
        .join("hall_request_assigner")
        .join("hall_request_assigner");

    let output = Command::new("sudo")
        .arg(cost_path)
        .arg("--input")
        .arg(json_str)
        .output()
        .await
        .expect("Failed to start algorithm");

    String::from_utf8_lossy(&output.stdout).into_owned()
}

pub async fn create_hall_request_json(wv: Vec<u8>) -> Option<String> {
    let wv_deser = world_view::serial::deserialize_worldview(&wv);


    let mut states = HashMap::new();
    for elev in wv_deser.elevator_containers.iter() {
        let key = elev.elevator_id.to_string();
        if elev.behaviour != ElevatorBehaviour::Error {
            states.insert(
            key,
            ElevatorState {
                behaviour: match elev.behaviour.clone() {
                    ElevatorBehaviour::DoorOpen => {
                        format!("doorOpen")
                    }
                    _ => {
                        format!("{:?}", elev.behaviour.clone()).to_lowercase()
                    }
                },
                floor: if (0..elev.num_floors).contains(&elev.last_floor_sensor) {
                    elev.last_floor_sensor as i32
                } else {
                    // TODO: Init floor er 255, bedre måte enn å sette til 2?
                    2
                },
                direction: format!("{:?}", elev.dirn.clone()).to_lowercase(),
                cabRequests: elev.cab_requests.clone(),
                },
            );
        }
    }

    if states.is_empty() {
        return None
    }
    let request = HallRequests {
        hallRequests: wv_deser.hall_request,
        states,
    };

    let s = serde_json::to_string_pretty(&request).expect("Failed to serialize");

    let mut file = File::create("hall_request.json").expect("Failed to create file");
    file.write_all(s.as_bytes()).expect("Failed to write to file");
    Some(s)
    // run_cost_algorithm(s.clone()).await
}
```

Innhald frå Rust-filer

Fil: elevator_pro_rebrand/src/manager/mod.rs

```
use std::collections::HashMap;
use tokio::{sync::{mpsc, watch}, time::sleep};
use crate::{config, world_view};

mod json_serial;



pub async fn start_manager(wv_watch_rx: watch::Receiver<Vec<u8>>, delegated_tasks_tx: mpsc::Sender<HashMap<u8, Vec<[bool; 2]>>>) {
    let mut wv = world_view::get_wv(wv_watch_rx.clone());
    loop {
        if world_view::update_wv(wv_watch_rx.clone(), &mut wv).await {
            if world_view::is_master(wv.clone()) {
                let _ = delegated_tasks_tx.send(get_elev_tasks(wv.clone()).await).await;
            }else {
                sleep(config::SLAVE_TIMEOUT).await;
            }
        }

        sleep(config::POLL_PERIOD).await;
    }
}



// async fn get_elev_tasks(wv: Vec<u8>) -> HashMap<u8, Vec<[bool; 2]>> {
//     let json_str = json_serial::create_hall_request_json(wv).await;
//     // println!("json_str: {}", json_str.clone());
//     if let Some(str) = json_str {
//         let json_cost_str = json_serial::run_cost_algorithm(str).await;
//         return serde_json::from_str(&json_cost_str).expect("Faild to deserialize_json_to_map");
//     }
//     return HashMap::new()
//     // println!("json_cost_str: {}", json_cost_str.clone());
// }


//Ditta fjerna Panicken, men vi krasjer enda npr vi starter under 0 etasje
async fn get_elev_tasks(wv: Vec<u8>) -> HashMap<u8, Vec<[bool; 2]>> {
    let json_str = json_serial::create_hall_request_json(wv).await;

    if let Some(str) = json_str {
        let json_cost_str = json_serial::run_cost_algorithm(str.clone()).await;

        if json_cost_str.trim().is_empty() {
            eprintln!(" run_cost_algorithm returnerte tom streng!, vi sendte {}", str);
            return HashMap::new();
        }
        return serde_json::from_str(&json_cost_str)
            .unwrap_or_else(|e| {
                eprintln!(" JSON-parsing feila: {}", e);
                HashMap::new()
            });
    }

    eprintln!(" create_hall_request_json returnerte None! string inn:");
    HashMap::new()
}
```

Innhald frå Rust-filer

Fil: elevator_pro_rebrand/src/network/tcp_network.rs

```
//! ## Håndterer TCP-logikk i systemet

use std::sync::atomic::{AtomicBool, Ordering};
use tokio::{io::{AsyncReadExt, AsyncWriteExt}, net::{TcpListener, TcpStream}, task::JoinHandle, sync::{mpsc, watch},
time::{sleep, Duration, Instant}};
use std::net::SocketAddr;
use crate::{config, print, network, ip_help_functions::{self}, world_view::{self, serial}};


/* __________ START PUBLIC FUNCTIONS __________ */

/// AtomicBool representing if you are master on the network.
///
/// The value is initialized as false
pub static IS_MASTER: AtomicBool = AtomicBool::new(false);


/// Handles the TCP listener
///
/// # Parameters
/// `socket_tx`: mpsc Sender on channel for sending newly connected slaves
///
/// # Return
/// The functions returns if any fatal errors occures
///
/// # Behavior
/// The function sets up a listener as soon as the system is online.
/// While the program is online, it accepts new connections on the listener, and sends the socket over `socket_tx`.
///
pub async fn listener_task(socket_tx: mpsc::Sender<(TcpStream, SocketAddr)>) {
    /* On first init. make sure the system is online so no errors occures while setting up the listener */
    while !network::read_network_status() {
        tokio::time::sleep(config::TCP_PERIOD).await;
    }

    let self_ip = format!("{}.{}", config::NETWORK_PREFIX, network::read_self_id());
    /* Bind the listener on port [config::PN_PORT] */
    let listener = match TcpListener::bind(format!("{}:{}", self_ip, config::PN_PORT)).await {
        Ok(l) => {
            print::ok(format!("System listening on {}:{}", self_ip, config::PN_PORT));
            l
        }
        Err(e) => {
            print::err(format!("Error while setting up TCP listener: {}", e));
            return;
        }
    };

    loop {
        /* Check if you are online */
        if network::read_network_status() {
            sleep(Duration::from_millis(100)).await;
            /* Accept new connections */
            match listener.accept().await {
                Ok((socket, addr)) => {
                    print::master(format!("{} connected to TCP", addr));
                    if socket_tx.send((socket, addr)).await.is_err() {
                        print::err("socker_rx is closed, returning".to_string());
                        break;
                    }
                }
                Err(e) => {
                    print::err(format!("Error while accepting slave connection: {}", e));
                }
            }
        } else {
            sleep(config::OFFLINE_PERIOD).await;
        }
    }
}

/// Function that handles TCP-connections in the system
///
/// # Parameters
/// `wv_watch_rx`: Reciever on watch the worldview is being sent on in the system
/// `remove_container_tx`: mpsc Sender used to notify worldview updater if a slave should be removed
/// `connection_to_master_failed`: Sender on mpsc channel signaling if connection to master has failed
/// `sent_tcp_container_tx`: mpsc Sender for notifying worldview updater what data has been sent to master
/// `container_tx`: mpsc Sender used pass recieved slave-messages to the worldview_updater
/// `socket_rx`: Reciever on mpsc channel recieving new TcpStreams and SocketAddress from the TCP listener
///
/// # Behavior
/// The function loops:
/// - Call and await [tcp_while_master].
/// - Call and await [tcp_while_slave].
///
/// # Note
/// - If the function is called without internet connection, it will not do anything before internet connection is back up again.
/// - The function is dependant on [listener_task] to be running for the master-behavior to work as excpected.
///
pub async fn tcp_handler(
    wv_watch_rx: watch::Receiver<Vec<u8>>,
    remove_container_tx: mpsc::Sender<u8>,
    container_tx: mpsc::Sender<Vec<u8>>,
    connection_to_master_failed_tx: mpsc::Sender<bool>,
    sent_tcp_container_tx: mpsc::Sender<Vec<u8>>,
    mut socket_rx: mpsc::Receiver<(TcpStream, SocketAddr)>
)
{
    while !network::read_network_status() {

    }
    let mut wv = world_view::get_wv(wv_watch_rx.clone());
    loop {
        IS_MASTER.store(true, Ordering::SeqCst);
                    tcp_while_master(&mut wv, wv_watch_rx.clone(), &mut socket_rx, remove_container_tx.clone(),
container_tx.clone()).await;

        //mista master
        IS_MASTER.store(false, Ordering::SeqCst);
                                tcp_while_slave(&mut wv, wv_watch_rx.clone(), connection_to_master_failed_tx.clone(),
sent_tcp_container_tx.clone()).await;

        //ny master
    }
}

/* __________ END PUBLIC FUNCTIONS __________ */









/* __________ START PRIVATE FUNCTIONS __________ */

/// Handles timeout on TCP connection at master, and reading from slave
struct TcpWatchdog {
    timeout: Duration,
}

impl TcpWatchdog {
    /// Starts a loop where reading from stream and checking for timeout runs asynchronously
    ///
    /// # Parameters
    /// `stream`: The TCP-stream to be read from
    /// `remove_container_tx`: mpsc Sender used to notify the worldview updater if a slave should be remover
    /// `container_tx`: mpsc Sender used to pass recieved slave-messages to the worldview_updater
    ///
    /// # Behavior
    /// The function loops:
    /// - Calculate time before a timeout occures
    /// - Asynchronously select between:
    ///     - Sending the data successfully recieved on the TCP stream over `container_tx`
    ///     - Sending the ID of the slave on `remove_container_tx` on timeout event
        async fn start_reading_from_slave(&self, mut stream: TcpStream, remove_container_tx: mpsc::Sender<u8>,
container_tx: mpsc::Sender<Vec<u8>>) {
        let mut last_success = Instant::now();
        loop {
            /* Calculate how long until timout occures */
            let remaining = self.timeout
                .checked_sub(last_success.elapsed())
                .unwrap_or(Duration::from_secs(0));

            /* Creates a sleep-future based on remaining time before timeout */
            let sleep_fut = sleep(remaining);
            tokio::pin!(sleep_fut);

            tokio::select! {
                /* Tries to read from stream */
                result = read_from_stream(remove_container_tx.clone(), &mut stream) => {
                    match result {
                        Some(msg) => {
                            let _ = container_tx.send(msg).await;
                            last_success = Instant::now()
                        }
                        None => {
                            break;
                        }
                    }
                }
                /* Triggers if no message is recieved within the timeout-duration */
                _ = &mut sleep_fut => {
                    print::err(format!("Timeout: No message recieved within: {:?}", self.timeout));
                    let id = ip_help_functions::ip2id(stream.peer_addr().expect("Peer has no IP?").ip());
                    print::info(format!("Closing stream to slave {}", id));
                    let _ = remove_container_tx.send(id).await;
                    close_tcp_stream(&mut stream).await;
                    break;
                }
            }
        }
    }
}


/// Function that handles TCP while you are master on the system
///
/// # Parameters
/// `wv`: A mutable refrence to the current serialized worldview
/// `wv_watch_rx`: Reciever on watch the worldview is being sent on in the system
/// `socket_rx`: Reciever on mpsc channel recieving new TcpStreams and SocketAddress from the TCP listener
/// `remove_container_tx`: mpsc Sender used to notify worldview updater if a slave should be removed
/// `container_tx`: mpsc Sender used pass recieved slave-messages to the worldview_updater
///
/// # Behavior
/// While the system is master on the network:
/// - Recieve new TcpStreams on `socket_rx`.
/// - If a new TcpStream is recieved, it starts [TcpWatchdog::start_reading_from_slave] on the stream
async fn tcp_while_master(wv: &mut Vec<u8>, wv_watch_rx: watch::Receiver<Vec<u8>>, socket_rx: &mut
mpsc::Receiver<(TcpStream, SocketAddr)>, remove_container_tx: mpsc::Sender<u8>, container_tx: mpsc::Sender<Vec<u8>>) {
    /* While you are master */
    while world_view::is_master(wv.clone()) {
        /* Check if you are online */
        if network::read_network_status() {
            /* Revieve TCP-streams to newly connected slaves */
            while let Ok((stream, addr)) = socket_rx.try_recv() {
                print::info(format!("New slave connected: {}", addr));

                let remove_container_tx_clone = remove_container_tx.clone();
                let container_tx_clone = container_tx.clone();
                let _slave_task: JoinHandle<()> = tokio::spawn(async move {
                    let tcp_watchdog = TcpWatchdog {
                        timeout: Duration::from_millis(config::TCP_TIMEOUT),
                    };
                    /* Start handling the slave. Also has watchdog function to detect timeouts on messages */
                    tcp_watchdog.start_reading_from_slave(stream, remove_container_tx_clone, container_tx_clone).await;
                });
                /* Make sure other tasks are able to run */
                tokio::task::yield_now().await;
            }
        }
        else {
            tokio::time::sleep(Duration::from_millis(100)).await;
        }
        world_view::update_wv(wv_watch_rx.clone(), wv).await;
    }
}

/// This function handles tcp connection while you are a slave on the system
///
/// # Parameters
/// `wv`: A mutable refrence to the current serialized worldview
/// `wv_watch_rx`: Reciever on watch the worldview is being sent on in the system
/// `connection_to_master_failed`: Sender on mpsc channel signaling if connection to master has failed
/// `sent_tcp_container_tx`: mpsc Sender for notifying worldview updater what data has been sent to master
///
///
/// # Behavior
/// The function tries to connect to the master.
/// While the system is a slave on the network and connection to the master is valid:
/// - Send TCP message to the master
/// - Check for new master on the system
async fn tcp_while_slave(wv: &mut Vec<u8>, wv_watch_rx: watch::Receiver<Vec<u8>>, connection_to_master_failed_tx: mpsc::Sender<bool>, sent_tcp_container_tx: mpsc::Sender<Vec<u8>>) {
    /* Try to connect with master over TCP */
    let mut master_accepted_tcp = false;
    let mut stream:Option<TcpStream> = None;
    if let Some(s) = connect_to_master(wv_watch_rx.clone(), connection_to_master_failed_tx.clone()).await {
        println!("Master accepted the TCP-connection");
        master_accepted_tcp = true;
        stream = Some(s);
    } else {
        println!("Master adid not accept the TCP-connection");
    }

    let mut prev_master: u8;
    let mut new_master = false;
    /* While you are slave and tcp-connection to master is good */
    while !world_view::is_master(wv.clone()) && master_accepted_tcp {
        /* Check if you are online */
        if network::read_network_status() {
            if let Some(ref mut s) = stream {
                /* Send TCP message to master */
                                send_tcp_message(connection_to_master_failed_tx.clone(), sent_tcp_container_tx.clone(), s,
wv.clone()).await;
                if new_master {
                    print::slave(format!("New master on the network"));
                    master_accepted_tcp = false;
                    let _ = sleep(config::SLAVE_TIMEOUT);
                }
                prev_master = wv[config::MASTER_IDX];
                world_view::update_wv(wv_watch_rx.clone(), wv).await;
                if prev_master != wv[config::MASTER_IDX] {
                    new_master = true;
                }
                tokio::time::sleep(config::TCP_PERIOD).await;
            }
        }
        else {
            let _ = sleep(config::SLAVE_TIMEOUT);
        }
    }
}



/// Attempts to connect to master over TCP
///
/// # Parameters
/// `wv_watch_rx`: Reciever on watch the worldview is being sent on in the system
/// `connection_to_master_failed`: Sender on mpsc channel signaling if connection to master has failed
///
/// # Return
/// `Some(TcpStream)`: Connection to master successfull, TcpStream is the stream to the master
/// `None`: Connection to master failed
///
/// # Behavior
/// The functions tries to connect to the current master, based on the master_id in the worldview.
/// If the connection is successfull, it returns the stream, otherwise it returns None.
/// If the connection failed, it sends a signal to the worldview updater over `connection_to_master_failed_tx` indicating
that the connection failed.
async fn connect_to_master(wv_watch_rx: watch::Receiver<Vec<u8>>, connection_to_master_failed_tx:
```

```
mpsc::Sender<bool>) -> Option<TcpStream> {
    let wv = world_view::get_wv(wv_watch_rx.clone());

    /* Check if we are online */
    if network::read_network_status() {
        let master_ip = format!("{}.{}:{}", config::NETWORK_PREFIX, wv[config::MASTER_IDX], config::PN_PORT);
        print::info(format!("Trying to connect to : {} in connect_to_master()", master_ip));

        /* Try to connect to master */
        match TcpStream::connect(&master_ip).await {
            Ok(stream) => {
                print::ok(format!("Connected to Master: {} i TCP_listener()", master_ip));
                // Klarte å koble til master, returner streamen
                Some(stream)
            }
            Err(e) => {
                print::err(format!("Failed to connect to master over tcp: {}", e));

                match connection_to_master_failed_tx.send(true).await {
                    Ok(_) => print::info("Notified that connection to master failed".to_string()),
                    Err(err) => print::err(format!("Error while sending message on connection_to_master_failed: {}", err)),
                }
                None
            }
        }
    } else {
        None
    }
}




/// ## Leser fra `stream`
///
/// Select mellom å lese melding fra slave og sende meldingen til `world_view_handler` og å avslutte streamen om du ikke er master

/// Function to read message from slave
///
/// # Parameters
/// `remove_container_tx`: mpsc Sender for channel used to indicate a slave should be removed at worldview updater
/// `stream`: the stream to read from
///
/// # Return
/// `Some(Vec<u8>)`: The serialized message if it was read succesfully
/// `None`: If reading from stream fails, or you become slave
///
/// # Behavior
/// The function reads from stream. It first reads a header (2 bytes) indicating the message length.
/// Based on the header it reads the message. If everything works without error, it returns the message.
/// The function also asynchronously checks for loss of master status, and returns None if that is the case.
///
async fn read_from_stream(remove_container_tx: mpsc::Sender<u8>, stream: &mut TcpStream) -> Option<Vec<u8>> {
    let mut len_buf = [0u8; 2];
    tokio::select! {
        result = stream.read_exact(&mut len_buf) => {
            match result {
                Ok(0) => {
                    print::info("Slave disconnected.".to_string());
                    let id = ip_help_functions::ip2id(stream.peer_addr().expect("Slave has no IP?").ip());
                    let _ =  remove_container_tx.send(id).await;
                    return None;
                }
                Ok(_) => {
                    let len = u16::from_be_bytes(len_buf) as usize;
                    let mut buffer = vec![0u8; len];

                    match stream.read_exact(&mut buffer).await {
                        Ok(0) => {
                            print::info("Slave disconnected".to_string());
                            let id = ip_help_functions::ip2id(stream.peer_addr().expect("Slave has no IP?").ip());
                            let _ =  remove_container_tx.send(id).await;
                            return None;
                        }
                        Ok(_) => return Some(buffer),
                        Err(e) => {
                            print::err(format!("Error while reading from stream: {}", e));
                            let id = ip_help_functions::ip2id(stream.peer_addr().expect("Slave has no IP?").ip());
                            let _ =  remove_container_tx.send(id).await;
                            return None;
                        }
                    }
                }
                Err(e) => {
                    print::err(format!("Error while reading from stream: {}", e));
                    let id = ip_help_functions::ip2id(stream.peer_addr().expect("Slave has no IP?").ip());
                    let _ =  remove_container_tx.send(id).await;
                    return None;
                }
            }
        }
        _ = async {
            while IS_MASTER.load(Ordering::SeqCst) {
                tokio::time::sleep(Duration::from_millis(50)).await;
            }
        } => {
            let id = ip_help_functions::ip2id(stream.peer_addr().expect("Peer has no IP?").ip());
            print::info(format!("Losing master status! Removing slave {}", id));
            let _ =  remove_container_tx.send(id).await;
            return None;
        }
    }
}

/// ### Sender egen elevator_container til master gjennom stream
/// Sender på format : `(lengde av container) as u16`, `container`
///
/// Function that sends tcp message to master
///
/// # Parameters
/// `connection_to_master_failed_tx`: mpsc Sender for signaling to worldview updater that connection to master failed
/// `sent_tcp_container_tx`: mpsc Sender for notifying worldview updater what data has been sent to master
/// `stream`: The TcpStream to the master
/// `wv`: The current worldview in serial state
///
/// # Behavior
/// The functions extracts the systems own elevatorcontainer from the worldview.
/// The function writes the following on the stream's transmission-buffer:
/// - Length of the message
/// - The message
/// After this, it flushes the stream, and sends the sent data over `ent_tcp_container_tx`. If writing to the stream fails, it signals on `connection_to_master_failed_tx`
async fn send_tcp_message(connection_to_master_failed_tx: mpsc::Sender<bool>, sent_tcp_container_tx: mpsc::Sender<Vec<u8>>, stream: &mut TcpStream, wv: Vec<u8>) {
    let self_elev_container = match world_view::extract_self_elevator_container(wv) {
        Some(container) => container,
        None => {
            print::warn(format!("Failed to extract self elevator container"));
            return;
        }
    };

    let self_elev_serialized = serial::serialize_elev_container(&self_elev_container);

    /* Find number of bytes in the data to be sent */
    let len = (self_elev_serialized.len() as u16).to_be_bytes();

    /* Send the message */
    if let Err(_) = stream.write_all(&len).await {
        let _ = connection_to_master_failed_tx.send(true).await;
    } else if let Err(_) = stream.write_all(&self_elev_serialized).await {
        let _ = connection_to_master_failed_tx.send(true).await;
    } else if let Err(_) = stream.flush().await {
        let _ = connection_to_master_failed_tx.send(true).await;
    } else {
        let _ = sent_tcp_container_tx.send(self_elev_serialized).await;
    }
}

/// Closes the provided TCP stream asynchronously, logging the result.
///
/// This function attempts to close the provided TCP stream by invoking the `shutdown` method on the stream asynchronously.
/// It also retrieves the local and peer addresses of the stream, printing them in the log messages. If the stream is
```

```rust
/// closed successfully, a info message is printed. If an error occurs during the process, an error message is logged.
///
/// ## Parameters
/// - `stream`: The TCP stream to close (mutable reference to `TcpStream`).
///
/// ## Logs
/// - On success: Logs an info message such as "TCP connection closed successfully: <local_addr> -> <peer_addr>".
/// - On error: Logs an error message such as "Failed to close TCP connection (<local_addr> -> <peer_addr>): <error>".
async fn close_tcp_stream(stream: &mut TcpStream) {
    /* Get local and peer address */
    let local_addr = stream.local_addr().map_or_else(
        |e| format!("Ukjent (Feil: {})", e),
        |addr| addr.to_string(),
    );
    let peer_addr = stream.peer_addr().map_or_else(
        |e| format!("Ukjent (Feil: {})", e),
        |addr| addr.to_string(),
    );

    /* Try to shutdown the stream */
    match stream.shutdown().await {
        Ok(_) => print::info(format!(
            "TCP-connection closed successfully: {} -> {}",
            local_addr, peer_addr
        )),
        Err(e) => print::err(format!(
            "Failed to close TCP-connection ({} -> {}): {}",
            local_addr, peer_addr, e
        )),
    }
}

/* __________ END PRIVATE FUNCTIONS __________ */
```

Innhald frå Rust-filer

Fil: elevator_pro_rebrand/src/network/udp_network.rs

```
//! ## Håndterer UDP-logikk i systemet

use crate::config;
use crate::network;
use crate::print;
use crate::world_view;

use std::net::SocketAddr;
use std::sync::atomic::Ordering;
use std::sync::OnceLock;
use std::sync::atomic::AtomicBool;
use std::thread::sleep;
use std::time::Duration;
use tokio::net::UdpSocket;
use socket2::{Domain, Socket, Type};
use tokio::sync::mpsc;
use std::borrow::Cow;
use tokio::sync::watch;

static UDP_TIMEOUT: OnceLock<AtomicBool> = OnceLock::new();

/// Returns AtomicBool indicating if UDP has timeout'd.
///
/// Initialized as false.
pub fn get_udp_timeout() -> &'static AtomicBool {
    UDP_TIMEOUT.get_or_init(|| AtomicBool::new(false))
}

// ### Starter og kjører udp-broadcaster
/// This function starts and runs the UDP-broadcaster
///
/// ## Parameters
/// `wv_watch_rx`: Rx on watch the worldview is being sent on in the system
///
/// ## Behavior
/// - Sets up a reusable socket on the udp-broadcast address
/// - Continously reads the latest worldview, if self is master on the network, it broadcasts the worldview.
///
/// ## Note
/// This function is permanently blocking, and should be called asynchronously
pub async fn start_udp_broadcaster(wv_watch_rx: watch::Receiver<Vec<u8>>) -> tokio::io::Result<()> {
    while !network::read_network_status() {

    }
    let mut prev_network_status = network::read_network_status();

    // Sett opp sockets
    let addr: &str = &format!("{}:{}", config::BC_ADDR, config::DUMMY_PORT);
    let addr2: &str = &format!("{}:0", config::BC_LISTEN_ADDR);
    let broadcast_addr: SocketAddr = addr.parse().expect("ugyldig adresse"); // UDP-broadcast adresse
    let socket_addr: SocketAddr = addr2.parse().expect("Ugyldig adresse");
    let socket = Socket::new(Domain::IPV4, Type::DGRAM, None)?;

    socket.set_nonblocking(true)?;
    socket.set_reuse_address(true)?;
    socket.set_broadcast(true)?;
    socket.bind(&socket_addr.into())?;
    let udp_socket = UdpSocket::from_std(socket.into())?;

    let mut wv = world_view::get_wv(wv_watch_rx.clone());
    loop{
        let wv_watch_rx_clone = wv_watch_rx.clone();
        world_view::update_wv(wv_watch_rx_clone, &mut wv).await;

        // Hvis du er master, broadcast worldview
        if network::read_self_id() == wv[config::MASTER_IDX] {
            //TODO: Lag bedre delay?
            sleep(config::UDP_PERIOD);
            let mesage = format!("{:?}{:?}", config::KEY_STR, wv).to_string();

            // Kun send hvis du har internett-tilkobling
            if network::read_network_status() {
                // Gi den tid til å lese nye wv fra udp tilfelle den var ute av internett lenge
                if !prev_network_status {
                    sleep(Duration::from_millis(500));
                    prev_network_status = true;
                }
                udp_socket.send_to(mesage.as_bytes(), &broadcast_addr).await?;
            }else {
                prev_network_status = false;
            }
        }
    }
}

// ### Starter og kjører udp-listener
/// Starts and runs the UDP-listener
///
/// ## Parameters
/// `wv_watch_rx`: Rx on watch the worldview is being sent on in the system
/// `udp_wv_tx`: mpsc sender used to update [local_network::update_wv_watch] about new worldviews recieved over UDP
///
/// ## Behaviour
/// - Sets up a reusable listener listening for udp-broadcasts
/// - Continously reads on the listener
/// - Checks for key-string on all recieved messages, making sure the message is from one of 'our' nodes.
/// - If the message is from the current master or a node with lower ID than the current master, it sends it on `udp_wv_tx`
///
/// ## Note
/// This function is permanently blocking, and should be called asynchronously
```

Innhald frå Rust-filer

```
pub async fn start_udp_listener(wv_watch_rx: watch::Receiver<Vec<u8>>, udp_wv_tx: mpsc::Sender<Vec<u8>>) ->
tokio::io::Result<()> {
    while !network::read_network_status() {

    }
    //Sett opp sockets
    let self_id = network::read_self_id();
    let broadcast_listen_addr = format!("{}:{}", config::BC_LISTEN_ADDR, config::DUMMY_PORT);
    let socket_addr: SocketAddr = broadcast_listen_addr.parse().expect("Ugyldig adresse");
    let socket_temp = Socket::new(Domain::IPV4, Type::DGRAM, None)?;

    socket_temp.set_nonblocking(true).expect("Failed to set non-blocking");
    socket_temp.set_reuse_address(true)?;
    socket_temp.set_broadcast(true)?;
    socket_temp.bind(&socket_addr.into())?;
    let socket = UdpSocket::from_std(socket_temp.into())?;
    let mut buf = [0; config::UDP_BUFFER];
    let mut read_wv: Vec<u8>;

    let mut message: Cow<'_, str>;
    let mut my_wv = world_view::get_wv(wv_watch_rx.clone());
    // Loop mottar og behandler udp-broadcaster
    loop {
        match socket.recv_from(&mut buf).await {
            Ok((len, _)) => {
                message = String::from_utf8_lossy(&buf[..len]);
                // println!("WV length: {:?}", len);
            }
            Err(e) => {
                // utils::print_err(format!("udp_broadcast.rs, udp_listener(): {}", e));
                return Err(e);
            }
        }

        // Verifiser at broadcasten var fra 'oss'
         if &message[1..config::KEY_STR.len()+1] == config::KEY_STR { //Plusser på en, siden serialiseringa av stringen
tar med '"'-tegnet
            let clean_message = &message[config::KEY_STR.len()+3..message.len()-1]; // Fjerner `"`
            read_wv = clean_message
            .split(", ") // Del opp på ", "
            .filter_map(|s| s.parse::<u8>().ok()) // Konverter til u8, ignorer feil
            .collect(); // Samle i Vec<u8>

            world_view::update_wv(wv_watch_rx.clone(), &mut my_wv).await;
            if read_wv[config::MASTER_IDX] != my_wv[config::MASTER_IDX] {
                // mulighet for debug print
            } else {
                // Betyr at du har fått UDP-fra nettverkets master -> Restart UDP watchdog
                get_udp_timeout().store(false, Ordering::SeqCst);
                // println!("Resetter UDP-watchdog");
            }
            // Hvis broadcast har lavere ID enn nettverkets tidligere master
            if my_wv[config::MASTER_IDX] >= read_wv[config::MASTER_IDX] {
                if !(self_id == read_wv[config::MASTER_IDX]) {
                    //Oppdater egen WV
                    my_wv = read_wv;
                    let _ = udp_wv_tx.send(my_wv.clone()).await;
                }
            }

        }
    }
}


/// Simple watchdog
///
/// # Parameters
/// `connection_to_master_failed_tx`: mpsc Sender that signals to the worldview updater that connection to the master has failed
///
/// # Behavior
/// The function stores true in an atomic bool, and sleeps for 1 second.
/// If the atomic bool is true when it wakes up, the watchdog has detected a timeout, as it is set false each time a UDP broadcast is recieved from the master.
/// If a timeout is detected, it signals that connection to master has failed.
pub async fn udp_watchdog(connection_to_master_failed_tx: mpsc::Sender<bool>) {
    while !network::read_network_status() {

    }
    loop {
        if get_udp_timeout().load(Ordering::SeqCst) == false {
            get_udp_timeout().store(true, Ordering::SeqCst);
            tokio::time::sleep(Duration::from_millis(1000)).await;
        }
        else {
            get_udp_timeout().store(false, Ordering::SeqCst); //resetter watchdogen
            print::warn("UDP-watchdog: Timeout".to_string());
            let _ = connection_to_master_failed_tx.send(true).await;
        }
    }
}
```

Innhald frå Rust-filer

Fil: elevator_pro_rebrand/src/network/mod.rs

```rust
pub mod tcp_network;
pub mod udp_network;
pub mod local_network;


use crate::{init, config, print, ip_help_functions, world_view, };

use tokio::sync::{mpsc, watch};
use std::sync::atomic::{Ordering, AtomicU8, AtomicBool};
use std::sync::OnceLock;
use local_ip_address::local_ip;
use std::net::IpAddr;



/// Returns the local IPv4 address of the machine as `IpAddr`.
///
/// If no local IPv4 address is found, returns `local_ip_address::Error`.
///
/// # Example
/// ```
/// use elevatorpro::network::local_network::get_self_ip;
///
/// match get_self_ip() {
///     Ok(ip) => println!("Local IP: {}", ip), // IP retrieval successful
///     Err(e) => println!("Failed to get IP: {:?}", e), // No local IP available
/// }
/// ```
pub fn get_self_ip() -> Result<IpAddr, local_ip_address::Error> {
    let ip = match local_ip() {
        Ok(ip) => {
            ip
        }
        Err(e) => {
            // print::warn(format!("Fant ikke IP i get_self_ip() -> Vi er offline: {}", e));
            return Err(e);
        }
    };
    Ok(ip)
}

/// Monitors the Ethernet connection status asynchronously.
///
/// This function continuously checks whether the device has a valid network connection.
/// It determines connectivity by verifying that the device's IP matches the expected network prefix.
/// The network status is stored in a shared atomic boolean [get_network_status()].
///
/// ## Behavior
/// - Retrieves the device's IP address using `utils::get_self_ip()`.
/// - Extracts the root IP using `utils::get_root_ip()` and compares it to `config::NETWORK_PREFIX`.
```

```
/// - Updates the network status (`true` if connected, `false` if disconnected).
/// - Prints status changes:
///   - `"Vi er online"` when connected.
///   - `"Vi er offline"` when disconnected.
///
/// ## Note
/// This function runs in an infinite loop and should be spawned as an asynchronous task.
///
/// ## Example
/// ```
/// use tokio;
/// # #[tokio::test]
/// # async fn test_watch_ethernet() {
/// tokio::spawn(async {
///     watch_ethernet().await;
/// });
/// # }
/// ```
pub async fn watch_ethernet(wv_watch_rx: watch::Receiver<Vec<u8>>, new_wv_after_offline_tx: mpsc::Sender<Vec<u8>>) {
    let mut last_net_status = false;
    let mut net_status;
    loop {
        let ip = get_self_ip();

        match ip {
            Ok(ip) => {
                if ip_help_functions::get_root_ip(ip) == config::NETWORK_PREFIX {
                    net_status = true;
                }
                else {
                    net_status = false
                }
            }
            Err(_) => {
                net_status = false
            }
        }

        if last_net_status != net_status {
            if net_status {
                let mut wv = world_view::get_wv(wv_watch_rx.clone());
                let self_elev = world_view::extract_self_elevator_container(wv.clone());
                wv = init::initialize_worldview(self_elev).await;
                let _ = new_wv_after_offline_tx.send(wv).await;
                print::ok("System is online".to_string());
            }
            else {
                print::warn("System is offline".to_string());
            }
            set_network_status(net_status);
            last_net_status = net_status;
```

```
        }
    }
}

static ONLINE: OnceLock<AtomicBool> = OnceLock::new();

/// Reads and returns a clone of the current network status
///
/// This function returns a copy of the network status the moment it was read.
/// that represents whether the system is online or offline.
///
/// # Returns
/// A bool`:
/// - `true` if the system is online.
/// - `false` if the system is offline.
///
/// # Note
/// - The initial value is `false` until explicitly changed.
/// - The returned value is only a clone of the atomic boolean's value at read-time. The function should be called every time you need to check the online-status
pub fn read_network_status() -> bool {
    ONLINE.get_or_init(|| AtomicBool::new(false)).load(Ordering::SeqCst)
}

/// This function sets the network status
fn set_network_status(status: bool) {
    ONLINE.get_or_init(|| AtomicBool::new(false)).store(status, Ordering::SeqCst);
}

/// Atomic bool storing self ID, standard inited as config::ERROR_ID
pub static SELF_ID: OnceLock<AtomicU8> = OnceLock::new();

/// Reads and returns a clone of the current sself ID
///
/// This function returns a copy of the self ID.
///
/// # Returns
/// u8: Your ID on the network
///
/// # Note
/// - The value is [config::ERROR_ID] if [watch_ethernet] is not running.
pub fn read_self_id() -> u8 {
    SELF_ID.get_or_init(|| AtomicU8::new(config::ERROR_ID)).load(Ordering::SeqCst)
}

/// This function sets your self ID
///
/// # Note
/// This function should not be used, as network ID is assigned automatically under initialisation
pub fn set_self_id(id: u8) {
    SELF_ID.get_or_init(|| AtomicU8::new(config::ERROR_ID)).store(id, Ordering::SeqCst);
}
```

Innhald frå Rust-filer

Innhald frå Rust-filer

Fil: elevator_pro_rebrand/src/network/local_network/update_wv.rs

```
use crate::world_view::{serial, ElevatorContainer, Dirn, ElevatorBehaviour};
use crate::{print, world_view, network};

use std::sync::atomic::Ordering;
use std::collections::HashMap;



/// Calls join_wv. See [join_wv]
/// TODO: drop denne funksjonen, la join_wv være join_wv_from_udp for å droppe unødvendige funksjoner
pub fn join_wv_from_udp(wv: &mut Vec<u8>, master_wv: Vec<u8>) -> bool {
    *wv = join_wv(wv.clone(), master_wv);
    true
}

/// Merges the local worldview with the master worldview received over UDP.
///
/// This function updates the local worldview (`my_wv`) by integrating relevant data
/// from `master_wv`. It ensures that the local elevator's status and tasks are synchronized
/// with the master worldview.
///
/// ## Arguments
/// * `my_wv` - A serialized `Vec<u8>` representing the local worldview.
/// * `master_wv` - A serialized `Vec<u8>` representing the worldview received over UDP.
///
/// ## Returns
/// A new serialized `Vec<u8>` representing the updated worldview.
///
/// ## Behavior
/// - If the local elevator exists in both worldviews, it updates its state in `master_wv`.
/// - Synchronizes `door_open`, `obstruction`, `last_floor_sensor`, and `motor_dir`.
/// - Updates `calls` and `tasks_status` with local data.
/// - Ensures that `tasks_status` retains only tasks present in `tasks`.
/// - If the local elevator is missing in `master_wv`, it is added to `master_wv`.
pub fn join_wv(mut my_wv: Vec<u8>, master_wv: Vec<u8>) -> Vec<u8> {
    let my_wv_deserialised = serial::deserialize_worldview(&my_wv);
    let mut master_wv_deserialised = serial::deserialize_worldview(&master_wv);


    let my_self_index = world_view::get_index_to_container(network::read_self_id() , my_wv);
    let master_self_index = world_view::get_index_to_container(network::read_self_id() , master_wv);


    if let (Some(i_org), Some(i_new)) = (my_self_index, master_self_index) {
        let my_view = &my_wv_deserialised.elevator_containers[i_org];
        let master_view = &mut master_wv_deserialised.elevator_containers[i_new];



        // Synchronize elevator status
        // master_view.status = my_view.status;
        master_view.dirn = my_view.dirn;
        master_view.behaviour = my_view.behaviour;
        master_view.obstruction = my_view.obstruction;
        master_view.last_floor_sensor = my_view.last_floor_sensor;

        // Update call buttons and task statuses
        // master_view.calls = my_view.calls.clone();
        master_view.unsent_hall_request = my_view.unsent_hall_request.clone();
        //Hvis anti-ny master (du blir master):
        if my_wv_deserialised.master_id > master_wv_deserialised.master_id {
            print::err("ERAHDSIAHD".to_string());
        }
        master_view.cab_requests = my_view.cab_requests.clone();

        /* Update task statuses */
        // let new_ids: HashSet<u16> = master_view.tasks.iter().map(|t| t.id).collect();
        // let old_ids: HashSet<u16> = master_view.tasks_status.iter().map(|t| t.id).collect();

        // // Add missing tasks from master's task list
        // for task in master_view.tasks.clone().iter() {
        //     if !old_ids.contains(&task.id) {
        //         master_view.tasks_status.push(task.clone());
        //     }
        // }
        // // Remove outdated tasks from task_status
        // master_view.tasks_status.retain(|t| new_ids.contains(&t.id));

        // Call buttons synchronization is handled through TCP reliability

    } else if let Some(i_org) = my_self_index {
        // If the local elevator is missing in master_wv, add it
        master_wv_deserialised.add_elev(my_wv_deserialised.elevator_containers[i_org].clone());
    }

    my_wv = serial::serialize_worldview(&master_wv_deserialised);
    //utils::print_info(format!("Oppdatert wv fra UDP: {:?}", my_wv));
    my_wv
}

/// ### 'Leaves' the network, removes all elevators that are not the current one
///
/// This function updates the local worldview by removing all elevators that do not
/// belong to the current entity, identified by `SELF_ID`.
///
/// The function first deserializes the worldview, removes all elevators that do not
/// have the correct `elevator_id`, updates the number of elevators, and sets the master
/// ID to `SELF_ID`. Then, the updated worldview is serialized back into `wv`.
///
/// ## Parameters
/// - `wv`: A mutable reference to a `Vec<u8>` representing the worldview.
///
/// ## Return Value
/// - Always returns `true` after the update.
///
/// ## Example
/// ```rust
/// let mut worldview = vec![/* some serialized data */];
/// abort_network(&mut worldview);
/// ```
pub fn abort_network(wv: &mut Vec<u8>) -> bool {
    let mut deserialized_wv = serial::deserialize_worldview(wv);
    deserialized_wv.elevator_containers.retain(|elevator| elevator.elevator_id == network::read_self_id());
    deserialized_wv.set_num_elev(deserialized_wv.elevator_containers.len() as u8);
    deserialized_wv.master_id = network::read_self_id();
    *wv = serial::serialize_worldview(&deserialized_wv);
    true
}

/// ### Updates the worldview based on a TCP message from a slave
///
/// This function processes a TCP message from a slave elevator, updating the local
/// worldview by adding the elevator if it doesn't already exist, or updating its
/// status and call buttons if it does.
///
/// The function first deserializes the TCP container and the current worldview.
/// It then checks if the elevator exists in the worldview and adds it if necessary.
/// After that, it updates the elevator's status and call buttons by calling appropriate
/// helper functions. Finally, it serializes the updated worldview and returns `true`.
/// If the elevator cannot be found in the worldview, an error message is printed and `false` is returned.
///
/// ## Parameters
/// - `wv`: A mutable reference to a `Vec<u8>` representing the worldview.
/// - `container`: A `Vec<u8>` containing the serialized data of the elevator's state.
///
/// ## Return Value
/// - Returns `true` if the update was successful, `false` if the elevator was not found in the worldview.
///
/// ## Example
/// ```
/// let mut worldview = vec![/* some serialized data */];
/// let container = vec![/* some serialized elevator data */];
/// join_wv_from_tcp_container(&mut worldview, container).await;
/// ```
pub async fn join_wv_from_tcp_container(wv: &mut Vec<u8>, container: Vec<u8>) -> bool {
    let deser_container = serial::deserialize_elev_container(&container);
    let mut deserialized_wv = serial::deserialize_worldview(&wv);

    // Hvis slaven ikke eksisterer, legg den til som den er
    if None == deserialized_wv.elevator_containers.iter().position(|x| x.elevator_id == deser_container.elevator_id) {
        deserialized_wv.add_elev(deser_container.clone());
    }

    let self_idx = world_view::get_index_to_container(deser_container.elevator_id,
serial::serialize_worldview(&deserialized_wv));
```

```
    if let Some(i) = self_idx {

        // Legg til slave sine sendte hall_request til worldview sin hall_request
        for (row1, row2) in deserialized_wv.hall_request.iter_mut().zip(deser_container.unsent_hall_request.iter()) {
            for (val1, val2) in row1.iter_mut().zip(row2.iter()) {
                if !*val1 && *val2 {
                    *val1 = true;
                }
            }
        }

        if world_view::is_master(wv.clone()) {
                                                    deserialized_wv.elevator_containers[i].unsent_hall_request = vec![[false; 2];
deserialized_wv.elevator_containers[i].num_floors as usize];
        }

        //Oppdater statuser
        deserialized_wv.elevator_containers[i].cab_requests = deser_container.cab_requests;
        deserialized_wv.elevator_containers[i].elevator_id = deser_container.elevator_id;
        deserialized_wv.elevator_containers[i].last_floor_sensor = deser_container.last_floor_sensor;
        deserialized_wv.elevator_containers[i].num_floors = deser_container.num_floors;
        deserialized_wv.elevator_containers[i].obstruction = deser_container.obstruction;
        deserialized_wv.elevator_containers[i].dirn = deser_container.dirn;
        deserialized_wv.elevator_containers[i].behaviour = deser_container.behaviour;
        // Master styrer task, ikke overskriv det med slaven sitt forrige WV

        //Fjern tatt hall_requests. TODO: bedre? gjør mer forståelig
        for (idx, [up, down]) in deserialized_wv.hall_request.iter_mut().enumerate() {
                                    if (deserialized_wv.elevator_containers[i].behaviour == ElevatorBehaviour::DoorOpen) &&
(deserialized_wv.elevator_containers[i].last_floor_sensor == (idx as u8)) {
                if deserialized_wv.elevator_containers[i].dirn == Dirn::Up {
                    *up = false;
                } else if deserialized_wv.elevator_containers[i].dirn == Dirn::Down {
                    *down = false;
                }
            }
        }

        // Oppdater cab_request backupen!
                                                        update_cab_request_backup(&mut deserialized_wv.cab_requests_backup,
deserialized_wv.elevator_containers[i].clone());

        *wv = serial::serialize_worldview(&deserialized_wv);
        return true;
    } else {
         //Hvis dette printes, finnes ikke slaven i worldview. I teorien umulig, ettersom slaven blir lagt til over hvis den ikke
allerede eksisterte
        print::cosmic_err("The elevator does not exist join_wv_from_tcp_conatiner()".to_string());
        return false;
    }
}
```

```
/// ### Removes a slave based on its ID
///
/// This function removes an elevator (slave) from the worldview by its ID.
/// It first deserializes the current worldview, removes the elevator container
/// with the specified ID, and then serializes the updated worldview back into
/// the `wv` parameter.
///
/// ## Parameters
/// - `wv`: A mutable reference to a `Vec<u8>` representing the current worldview.
/// - `id`: The ID of the elevator (slave) to be removed.
///
/// ## Return Value
/// - Returns `true` if the removal was successful. In the current implementation,
///   it always returns `true` after the removal, as long as no errors occur during
///   the deserialization and serialization processes.
///
/// ## Example
/// ```rust
/// let mut worldview = vec![/* some serialized data */];
/// let elevator_id = 2;
/// remove_container(&mut worldview, elevator_id);
/// ```
pub fn remove_container(wv: &mut Vec<u8>, id: u8) -> bool {
    let mut deserialized_wv = serial::deserialize_worldview(&wv);
    deserialized_wv.remove_elev(id);
    *wv = serial::serialize_worldview(&deserialized_wv);
    true
}



/// ### Updates local call buttons and task statuses after they are sent over TCP to the master
///
/// This function processes the tasks and call buttons that have been sent to the master over TCP.
/// It removes the updated tasks and sent call buttons from the local worldview, ensuring that the
/// local state reflects the changes made by the master.
///
/// ## Parameters
/// - `wv`: A mutable reference to a `Vec<u8>` representing the current worldview.
/// - `tcp_container`: A vector containing the serialized data of the elevator container
///   that was sent over TCP, including the tasks' status and call buttons.
///
/// ## Return Value
/// - Returns `true` if the update was successful and the worldview was modified.
/// - Returns `false` if the elevator does not exist in the worldview.
///
/// ## Example
/// ```rust
/// let mut worldview = vec![/* some serialized data */];
/// let tcp_container = vec![/* some serialized container data */];
/// clear_from_sent_tcp(&mut worldview, tcp_container);
```

```
/// ```
pub fn clear_from_sent_tcp(wv: &mut Vec<u8>, tcp_container: Vec<u8>) -> bool {
    let mut deserialized_wv = serial::deserialize_worldview(&wv);
    let self_idx = world_view::get_index_to_container(network::read_self_id() , wv.clone());
    let tcp_container_des = serial::deserialize_elev_container(&tcp_container);

    // Lagre task-IDen til alle sendte tasks.
    // let tasks_ids: HashSet<u16> = tcp_container_des
    //     .tasks_status
    //     .iter()
    //     .map(|t| t.id)
    //     .collect();

    if let Some(i) = self_idx {
        /*_____ Fjern Tasks som master har oppdatert _____ */
        // deserialized_wv.elevator_containers[i].tasks_status.retain(|t| tasks_ids.contains(&t.id));
        /*_____ Fjern sendte Hall request _____ */

        for (row1, row2) in deserialized_wv.elevator_containers[i].unsent_hall_request
                                                .iter_mut().zip(tcp_container_des.unsent_hall_request.iter()) {
            for (val1, val2) in row1.iter_mut().zip(row2.iter()) {
                if *val1 && *val2 {
                    *val1 = false;
                }
            }
        }


        *wv = serial::serialize_worldview(&deserialized_wv);
        return true;
    } else {
        print::cosmic_err("The elevator does not exist clear_sent_container_stuff()".to_string());
        return false;
    }
}

/// This function allocates tasks from the given map to the corresponding elevator_container's tasks vector
///
/// # Parameters
/// `wv`: A mutable reference to a serialized worldview
///
/// # Behavior
/// - Iterates through every elevator_container in the worldview
/// - If any tasks in the map matches the elevators ID, it sets the elevators tasks equal to the map's tasks
///
/// # Return
/// true
///
pub fn distribute_tasks(wv: &mut Vec<u8>, map: HashMap<u8, Vec<[bool; 2]>>) -> bool {
    let mut wv_deser = world_view::serial::deserialize_worldview(&wv.clone());

    for elev in wv_deser.elevator_containers.iter_mut() {
```

```
        if let Some(tasks) = map.get(&elev.elevator_id) {
            elev.tasks = tasks.clone();
        }
    }

    *wv = world_view::serial::serialize_worldview(&wv_deser);

    true
}



/// Updates states to the elevator in wv with same ID as container
pub fn update_elev_states(wv: &mut Vec<u8>, container: Vec<u8>) -> bool {
    let mut wv_deser = world_view::serial::deserialize_worldview(&wv.clone());
    let container_deser = world_view::serial::deserialize_elev_container(&container);

    let idx = world_view::get_index_to_container(container_deser.elevator_id, wv.clone());

    if let Some(i) = idx {
        wv_deser.elevator_containers[i].cab_requests = container_deser.cab_requests;
        wv_deser.elevator_containers[i].dirn = container_deser.dirn;
        wv_deser.elevator_containers[i].obstruction = container_deser.obstruction;
        wv_deser.elevator_containers[i].behaviour = container_deser.behaviour;
        wv_deser.elevator_containers[i].last_floor_sensor = container_deser.last_floor_sensor;
        wv_deser.elevator_containers[i].unsent_hall_request = container_deser.unsent_hall_request;
    }

    *wv = world_view::serial::serialize_worldview(&wv_deser);
    true
}

/// Updates the backup hashmap for cab_requests, så they are remembered on the network in the case of power loss on
a node
///
/// ## Parameters
/// `backup`: A mutable reference to the backup hashmap in the worldview
/// `container`: The new ElevatorContainer recieved
///
/// ## Behaviour
/// Insert the container's cab_requests in key: container.elevator_id. If no old keys matches the id, a new entry is added.
fn update_cab_request_backup(backup: &mut HashMap<u8, Vec<bool>>, container: ElevatorContainer) {
    backup.insert(container.elevator_id, container.cab_requests);
}





/// Merges local worldview with networks worldview after being offline
///
/// # Parameters
/// `my_wv`: Mutable reference to the local worldview
/// `read_wv`: Reference to the networks worldview
pub fn merge_wv_after_offline(my_wv: &mut Vec<u8>, read_wv: &Vec<u8>) {
    let my_wv_deser = world_view::serial::deserialize_worldview(&my_wv);
    let mut read_wv_deser = world_view::serial::deserialize_worldview(&read_wv);

    /* If you become the new master on the system */
    if my_wv_deser.master_id < read_wv_deser.master_id {
        read_wv_deser.hall_request = merge_hall_requests(&read_wv_deser.hall_request, &my_wv_deser.hall_request);
        read_wv_deser.master_id = my_wv_deser.master_id;
        let my_wv_elevs: Vec<ElevatorContainer> = my_wv_deser.elevator_containers;

        /* Map the IDs in the networks worldview */
        let existing_ids: std::collections::HashSet<u8> = read_wv_deser
            .elevator_containers
            .iter()
            .map(|e| e.elevator_id)
            .collect();

        /* Add elevators you had which the network didnt know about (yourself) */
        for elev in my_wv_elevs {
            if !existing_ids.contains(&elev.elevator_id) {
                read_wv_deser.elevator_containers.push(elev);
            }
        }

    } else {
        read_wv_deser.hall_request = merge_hall_requests(&read_wv_deser.hall_request, &my_wv_deser.hall_request);
    }

    *my_wv = world_view::serial::serialize_worldview(&read_wv_deser);
}

/// Function to merge hall requests
///
/// # Parameters
/// `hall_req_1`: Reference to one hall request vector
/// `hall_req_2`: Reference to other hall request vector
///
/// # Return
/// The merged hall request vector
///
/// # Behavior
/// The function merges the requests by performing an element-wise OR operation on all indexes.
/// If one vector is longer than the other, the shorter one is treated as if it had all extra values set to false.
///
/// # Example
/// ```
/// use elevatorpro::world_view::world_view_update::merge_hall_requests;
///
/// let hall_req_1 = vec![[true, false], [false, false]];
/// let hall_req_2 = vec![[false, true], [false, true]];
/// let merged_vec = merge_hall_requests(&hall_req_1, &hall_req_2);
```

```
///
/// assert_eq!(merged_vec, vec![[true, true], [false, true]]);
///
///
/// let hall_req_3 = vec![[true, false], [false, false], [true, false]];
/// let merged_vec_2 = merge_hall_requests(&hall_req_3, &merged_vec);
///
/// assert_eq!(merged_vec_2, vec![[true, true], [false, true], [true, false]]);
///
/// ```
///
fn merge_hall_requests(hall_req_1: &Vec<[bool; 2]>, hall_req_2: &Vec<[bool; 2]>) -> Vec<[bool; 2]> {
    let mut merged_hall_req = hall_req_1.clone();
    //Basically en bitwise OR på begge viewene sin hall_request
    merged_hall_req
        .iter_mut()
        .zip(hall_req_2)
        .for_each(|(read, my)| {
            read[0] |= my[0];
            read[1] |= my[1];
        });

    // Hvis gamle array er lengre (din heis har fler etasjer): utvid
    if hall_req_2.len() > hall_req_1.len() {
        merged_hall_req
            .extend_from_slice(&hall_req_2[hall_req_1.len()..]);
    }

    merged_hall_req
}
```

Innhald frå Rust-filer

Fil: elevator_pro_rebrand/src/network/local_network/mod.rs

```rust
//! Handles messages on internal channels regarding changes in worldview
mod update_wv;

use crate::print;

// use crate::manager::task_allocator::Task;
use update_wv::{
    join_wv_from_udp,
    abort_network,
    join_wv_from_tcp_container,
    remove_container,
    clear_from_sent_tcp,
    distribute_tasks,
    update_elev_states,
    merge_wv_after_offline,
};
use crate::world_view::{self, serial};

use tokio::sync::{mpsc, watch};
use std::collections::HashMap;





/// The function that updates the worldview watch.
///
/// # Note
/// It is **critical** that this function is run. This is the "heart" of the local system,
/// and is responsible in updating the worldview based on information recieved form other parts of the program.
#[allow(non_snake_case)]
pub async fn update_wv_watch(mut mpsc_rxs: MpscRxs, worldview_watch_tx: watch::Sender<Vec<u8>>, mut worldview_serialised: Vec<u8>) {
    println!("Starter update_wv");
    let _ = worldview_watch_tx.send(worldview_serialised.clone());

    let mut wv_edited_I = false;
    let mut master_container_updated_I = false;

    let (master_container_tx, mut master_container_rx) = mpsc::channel::<Vec<u8>>(100);
    loop {

/* CHANNELS SLAVE MAINLY RECIEVES ON */
        /*_____Update worldview based on information send on TCP_____ */
        match mpsc_rxs.sent_tcp_container.try_recv() {
            Ok(msg) => {
                wv_edited_I = clear_from_sent_tcp(&mut worldview_serialised, msg);
            },
            Err(_) => {},
        }
        /*_____Update worldview based on worldviews recieved on UDP_____ */
        match mpsc_rxs.udp_wv.try_recv() {
            Ok(master_wv) => {
                wv_edited_I = join_wv_from_udp(&mut worldview_serialised, master_wv);
            },
            Err(_) => {},
        }
        /*_____Update worldview when tcp to master has failed_____ */
        match mpsc_rxs.connection_to_master_failed.try_recv() {
            Ok(_) => {
                wv_edited_I = abort_network(&mut worldview_serialised);
            },
            Err(_) => {},
        }


/* CHANNELS MASTER MAINLY RECIEVES ON */
     /*_____Update worldview based on message from master (simulated TCP message, so the master treats its own elevator as a slave)_____*/
        match master_container_rx.try_recv() {
            Ok(container) => {
                wv_edited_I = join_wv_from_tcp_container(&mut worldview_serialised, container.clone()).await;
            },
            Err(_) => {},
        }
        /*_____Update worldview based on message from slave_____*/
        match mpsc_rxs.container.try_recv() {
            Ok(container) => {
                wv_edited_I = join_wv_from_tcp_container(&mut worldview_serialised, container.clone()).await;
            },
            Err(_) => {},
        }
        /*_____Update worldview when a slave should be removed_____ */
        match mpsc_rxs.remove_container.try_recv() {
            Ok(id) => {
                wv_edited_I = remove_container(&mut worldview_serialised, id);
            },
            Err(_) => {},
        }
        /*_____Update worldview when new tasks has been given_____ */
        match mpsc_rxs.delegated_tasks.try_recv() {
            Ok(map) => {
                wv_edited_I = distribute_tasks(&mut worldview_serialised, map);
            },
            Err(_) => {},
        }


/* CHANNELS MASTER AND SLAVE RECIEVES ON */
        /*____Update worldview based on changes in the local elevator_____ */
        match mpsc_rxs.elevator_states.try_recv() {
            Ok(container) => {
                wv_edited_I = update_elev_states(&mut worldview_serialised, container);
                master_container_updated_I = world_view::is_master(worldview_serialised.clone());
            },
            Err(_) => {},
        }
        /*_____Update worldview after you reconeccted to internet  */
        match mpsc_rxs.new_wv_after_offline.try_recv() {
            Ok(read_wv) => {
                merge_wv_after_offline(&mut worldview_serialised, &read_wv);
                let _ = worldview_watch_tx.send(worldview_serialised.clone());
            },
            Err(_) => {},
        }



        /*_____If master container has changed, send the container on master_container_tx_____ */
        if master_container_updated_I {
            if let Some(container) = world_view::extract_self_elevator_container(worldview_serialised.clone()) {
                let _ = master_container_tx.send(serial::serialize_elev_container(&container)).await;
            } else {
                print::warn(format!("Failed to extract self elevator container  skipping update"));
            }
            master_container_updated_I = false;
        }

        /* UPDATE WORLDVIEW WATCH */
        if wv_edited_I {
            let _ = worldview_watch_tx.send(worldview_serialised.clone());
            wv_edited_I = false;
        }
    }
}
```

```
// --- MPSC-KANALAR ---
```

```
/// Struct containing multiple MPSC (multi-producer, single-consumer) sender channels.
/// These channels are primarely used to send data to the task updating the local worldview.
#[allow(missing_docs)]
#[derive(Clone)]
pub struct MpscTxs {
    /// Sends a UDP worldview packet.
    pub udp_wv: mpsc::Sender<Vec<u8>>,
    /// Notifies if the TCP connection to the master has failed.
    pub connection_to_master_failed: mpsc::Sender<bool>,
    /// Sends elevator containers recieved from slaves on TCP.
    pub container: mpsc::Sender<Vec<u8>>,
    /// Requests the removal of a container by ID.
    pub remove_container: mpsc::Sender<u8>,
    /// Sends a TCP container message that has been transmitted to the master.
    pub sent_tcp_container: mpsc::Sender<Vec<u8>>,
    /// Additional buffered channels for various data streams.
    // pub pending_tasks: mpsc::Sender<Vec<Task>>,
    pub delegated_tasks: mpsc::Sender<HashMap<u8, Vec<[bool; 2]>>>,
    pub elevator_states: mpsc::Sender<Vec<u8>>,
    pub new_wv_after_offline: mpsc::Sender<Vec<u8>>,
    pub mpsc_buffer_ch6: mpsc::Sender<Vec<u8>>,
    pub mpsc_buffer_ch7: mpsc::Sender<Vec<u8>>,
    pub mpsc_buffer_ch8: mpsc::Sender<Vec<u8>>,
    pub mpsc_buffer_ch9: mpsc::Sender<Vec<u8>>,
}

/// Struct containing multiple MPSC (multi-producer, single-consumer) receiver channels.
/// These channels are used to receive data from different parts of the system.
#[allow(missing_docs)]
pub struct MpscRxs {
    /// Receives a UDP worldview packet.
    pub udp_wv: mpsc::Receiver<Vec<u8>>,
    /// Receives a notification if the TCP connection to the master has failed.
    pub connection_to_master_failed: mpsc::Receiver<bool>,
    /// Receives elevator containers recieved from slaves on TCP.
    pub container: mpsc::Receiver<Vec<u8>>,
    /// Receives requests to remove a container by ID.
    pub remove_container: mpsc::Receiver<u8>,
    /// Receives TCP container messages that have been transmitted.
    pub sent_tcp_container: mpsc::Receiver<Vec<u8>>,
    /// Additional buffered channels for various data streams.
    // pub pending_tasks: mpsc::Receiver<Vec<Task>>,
    pub delegated_tasks: mpsc::Receiver<HashMap<u8, Vec<[bool; 2]>>>,
    pub elevator_states: mpsc::Receiver<Vec<u8>>,
    pub new_wv_after_offline: mpsc::Receiver<Vec<u8>>,
    pub mpsc_buffer_ch6: mpsc::Receiver<Vec<u8>>,
    pub mpsc_buffer_ch7: mpsc::Receiver<Vec<u8>>,
    pub mpsc_buffer_ch8: mpsc::Receiver<Vec<u8>>,
    pub mpsc_buffer_ch9: mpsc::Receiver<Vec<u8>>,
}

/// Struct that combines MPSC senders and receivers into a single entity.
```

Innhald frå Rust-filer

```
pub struct Mpscs {
    /// Contains all sender channels.
    pub txs: MpscTxs,
    /// Contains all receiver channels.
    pub rxs: MpscRxs,
}

impl Mpscs {
    /// Creates a new `Mpscs` instance with initialized channels.
    pub fn new() -> Self {
        let (tx_udp, rx_udp) = mpsc::channel(300);
        let (tx_connection_to_master_failed, rx_connection_to_master_failed) = mpsc::channel(300);
        let (tx_container, rx_container) = mpsc::channel(300);
        let (tx_remove_container, rx_remove_container) = mpsc::channel(300);
        let (tx_sent_tcp_container, rx_sent_tcp_container) = mpsc::channel(300);
        // let (tx_new_task, rx_new_task) = mpsc::channel(300);
        // let (tx_pending_tasks, rx_pending_tasks) = mpsc::channel(300);
        let (tx_buf3, rx_buf3) = mpsc::channel(300);
        let (tx_buf4, rx_buf4) = mpsc::channel(300);
        let (tx_buf5, rx_buf5) = mpsc::channel(300);
        let (tx_buf6, rx_buf6) = mpsc::channel(300);
        let (tx_buf7, rx_buf7) = mpsc::channel(300);
        let (tx_buf8, rx_buf8) = mpsc::channel(300);
        let (tx_buf9, rx_buf9) = mpsc::channel(300);

        Mpscs {
            txs: MpscTxs {
                udp_wv: tx_udp,
                connection_to_master_failed: tx_connection_to_master_failed,
                container: tx_container,
                remove_container: tx_remove_container,
                sent_tcp_container: tx_sent_tcp_container,
                delegated_tasks: tx_buf3,
                elevator_states: tx_buf4,
                new_wv_after_offline: tx_buf5,
                mpsc_buffer_ch6: tx_buf6,
                mpsc_buffer_ch7: tx_buf7,
                mpsc_buffer_ch8: tx_buf8,
                mpsc_buffer_ch9: tx_buf9,
            },
            rxs: MpscRxs {
                udp_wv: rx_udp,
                connection_to_master_failed: rx_connection_to_master_failed,
                container: rx_container,
                remove_container: rx_remove_container,
                sent_tcp_container: rx_sent_tcp_container,
                delegated_tasks: rx_buf3,
                elevator_states: rx_buf4,
                new_wv_after_offline: rx_buf5,
                mpsc_buffer_ch6: rx_buf6,
                mpsc_buffer_ch7: rx_buf7,
                mpsc_buffer_ch8: rx_buf8,
                mpsc_buffer_ch9: rx_buf9,
            },
        }
    }
}


// --- WATCH-KANALER ---
/// Struct containing watch senders for state updates.
#[derive(Clone)]
pub struct WatchTxs {
    /// Sender for the `wv` channel, transmitting a vector of bytes.
    pub wv: watch::Sender<Vec<u8>>,
    // Sender for the `elev_task` channel, transmitting a list of tasks.
    // pub elev_task: watch::Sender<Vec<Task>>,
}

/// Struct containing watch receivers for listening to state updates.
#[derive(Clone)]
pub struct WatchRxs {
    /// Receiver for the `wv` channel, listening to a vector of bytes.
    pub wv: watch::Receiver<Vec<u8>>,
    // Receiver for the `elev_task` channel, listening to a list of tasks.
    // pub elev_task: watch::Receiver<Vec<Task>>,
}


/// Struct encapsulating both watch senders (`WatchTxs`) and receivers (`WatchRxs`).
#[derive(Clone)]
pub struct Watches {
    /// Transmitters for watch channels.
    pub txs: WatchTxs,
    /// Receivers for watch channels.
    pub rxs: WatchRxs,
}

impl Watches {
    /// Creates a new `Watches` instance with initialized watch channels.
    ///
    /// # Returns
    /// A `Watches` instance containing both senders and receivers.
    pub fn new() -> Self {
        let (wv_tx, wv_rx) = watch::channel(Vec::<u8>::new());
        // let (tx1, rx1) = watch::channel(Vec::new());

        Watches {
            txs: WatchTxs {
                wv: wv_tx,
                // elev_task: tx1,
            },
            rxs: WatchRxs {
                wv: wv_rx,
                // elev_task: rx1,
            },
        }
    }
}
```