Innhald frå Rust-filer

Fil: elevator_pro_rebrand\src\backup.rs

```
//! #  NOT part of the final solution  Legacy backup module
//!
//! **This module is NOT part of the final distributed system solution.**
//!
//! It was originally developed as an **concept for local fault tolerance**,
//! where a backup process would start automatically in a separate terminal if the main
//! elevator program crashed. This idea was **inspired by the fault tolerance mechanisms
//! presented in the real-time lab exercises** in TTK4145.
//!
//! ## Intended Failover Behavior (Not Active in Final Design):
//! - To **automatically restart** the elevator program locally in case of crashes.
//! - To allow the elevator to **serve pending tasks while offline**,
//!   even without reconnecting to the network.
//! - To eventually **rejoin the network** and synchronize with the system if a connection was restored.
//!
//! ##  Why is it not part of our solution?
//! After discussions with course assistants and a better understanding of the assignment,
//! it became clear that:
//! - The project aims to implement **a distributed system**, not local persistence or replication.
//! - A local failover process like this is conceptually similar to **writing to a file and reloading**,
//!   which is **explicitly not the intended direction** of the assignment.
//! - All call redundancy and recovery should happen through the **shared synchronized worldview**,
//!   not through isolated local state or takeover logic.
//!
//! As a result, the failover behavior was disabled (e.g., by using high takeover timeouts),
//! and this module now functions purely as a **GUI client**:
//! - Connects to the master
//! - Receives `WorldView` updates
//! - Visualizes elevator state and network status using a colorized print
//!
//! ##  Summary:
//! - This is a **separate visualization tool**, _not part of the distributed control logic_.
//! - It remains in the codebase as a helpful debug utility, but should not be considered a part of the system design.
//!
//! ##  Note:
//! In industrial applications, local crash recovery _might_ be useful,
//! especially to avoid reinitializing the elevator in a potentially unstable state.
//! For example, if a bug caused a crash, restarting **at the same point** could lead to
//! an immediate second crash. A clean backup process, starting with the previous tasks,
//! can offer a more controlled re-entry.
//!
//! However, this type of resilience mechanism falls outside the scope and intention
//! of this assignment, which emphasizes **distributed coordination and recovery**
//! via the networked `WorldView`, not local persistence or reboot logic.

use std::env;
use std::net::ToSocketAddrs;
use std::process::Command;
use std::sync::atomic::{AtomicBool, Ordering};
use std::io::{self, Write};
```

Innhald frå Rust-filer

```rust
use socket2::{Socket, Domain, Type, Protocol};
use tokio::net::{TcpListener, TcpStream};
use tokio::io::{AsyncReadExt, AsyncWriteExt};
use tokio::sync::watch;
use tokio::time::{sleep, timeout};
use serde::{Serialize, Deserialize};
use crate::network::ConnectionStatus;
use crate::world_view::{ WorldView, serialize};
use crate::{config, init, network, world_view};
use crate::print;


/// Struct representing the data sent from the main process to the backup client.
///
/// It contains two components:
/// - `worldview`: The current `WorldView` of the system, used for visualization and potential local control.
/// - `network_status`: The latest known network status (internet and elevator mesh).
///
/// This payload is serialized and transmitted over TCP to keep the backup client synchronized
/// with the live system state.
///
#[derive(Serialize, Deserialize, Clone, Debug)]
struct BackupPayload {
    pub worldview: world_view::WorldView,
    pub network_status: ConnectionStatus,
}

/// Atomic flag to ensure that the backup terminal is only launched once.
///
/// Prevents spawning multiple backup clients simultaneously. Once set to `true`,
/// repeated calls to `start_backup_terminal()` will have no effec
static BACKUP_STARTED: AtomicBool = AtomicBool::new(false);

/// Creates a non-blocking TCP listener on the specified port, with address reuse enabled.
///
/// This helper sets up a low-level socket bound to `localhost:<port>`, configured
/// for asynchronous operation and reuse of the address.
///
/// # Parameters
/// - `port`: The TCP port number to bind to.
///
/// # Returns
/// A `TcpListener` ready for accepting incoming connections.
///
/// # Panics
/// This function will panic if:
/// - The address cannot be resolved.
/// - No valid IPv4 address is found.
/// - Socket creation or binding fails.
fn create_reusable_listener(
    port: u16
) -> TcpListener {
    let addr_str = format!("localhost:{}", port);
    let addr_iter = addr_str
        .to_socket_addrs()
        .expect("Klarte ikkje resolve 'localhost'");

    let addr = addr_iter
        .filter(|a| a.is_ipv4())
        .next()
        .expect("Fann ingen IPv4-adresse for localhost");
    let socket = Socket::new(Domain::IPV4, Type::STREAM, Some(Protocol::TCP))
        .expect("Couldnt create socket");
    socket.set_nonblocking(true)
        .expect("Couldnt set non blocking");
    socket.set_reuse_address(true)
        .expect("Couldnt set reuse_address");
    socket.bind(&addr.into())
        .expect("Couldnt bind the socket");
    socket.listen(128)
        .expect("Couldnt listen on the socket");
    TcpListener::from_std(socket.into())
        .expect("Couldnt create TcpListener")
}

/// Launches a new terminal window and starts the program in backup mode.
///
/// Uses the current binary path and appends the `backup` argument, causing
/// the program to run as a backup client.
///
/// This function checks the `BACKUP_STARTED` flag to ensure only one
/// backup process is started.
///
/// # Notes
/// - Only supported on Unix-like systems using `gnome-terminal`.
/// - Has no effect if backup is already running.
fn start_backup_terminal() {
    if !BACKUP_STARTED.load(Ordering::SeqCst) {
        let current_exe = env::current_exe().expect("Couldnt extract the executable");
        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);
    }
}

/// Continuously sends serialized `BackupPayload` updates to a connected backup client.
///
/// This task runs on the backup-server side. It listens on a watch channel
/// for updated payloads and transmits them to the connected client over TCP.
```

```
///
/// # Parameters
/// - `stream`: The TCP connection to the backup client.
/// - `rx`: A `watch::Receiver` for updated `BackupPayload` values.
///
/// # Behavior
/// - If sending fails, a warning is printed and the backup terminal is relaunched after delay.
/// - The loop exits after failure, assuming a new client will reconnect.
async fn handle_backup_client(
    mut stream: TcpStream,
    rx: watch::Receiver<BackupPayload>
) {
    loop {
        let payload = rx.borrow().clone();
        let serialized = serialize(&payload);

        if let Err(e) = stream.write_all(&serialized).await {
            print::err(format!("Backup send error: {}", e));
            print::warn(format!("Prøver igjen om {:?}", config::BACKUP_TIMEOUT));
            sleep(config::BACKUP_TIMEOUT).await;
            BACKUP_STARTED.store(false, Ordering::SeqCst);
            start_backup_terminal();
            break;
        }

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


/// Starts the backup server, listening for incoming backup clients and
/// transmitting the current system state (`WorldView`) and network status.
///
/// # Parameters
/// - `wv_watch_rx`: Watch receiver for current `WorldView`.
/// - `network_watch_rx`: Watch receiver for current `ConnectionStatus`.
///
/// # Behavior
/// - Spawns a TCP listener to accept connections from a backup client.
/// - On connection, launches a handler that sends periodic `BackupPayload` updates.
/// - Spawns a task to continuously refresh the payload with the latest worldview and network status.
///
/// # Notes
/// - This function is blocking and must be run as an asynchronous task.
/// - It starts the backup terminal once at initialization.
/// - Failures to send payloads are printed but do not crash the server.
pub async fn start_backup_server(
    wv_watch_rx: watch::Receiver<WorldView>,
    network_watch_rx: watch::Receiver<network::ConnectionStatus>,
) {
    println!("Backup-server starting...");
```

```rust
    let listener = create_reusable_listener(config::BCU_PORT);
    let wv = world_view::get_wv(wv_watch_rx.clone());
    let initial_payload = BackupPayload {
        worldview: wv.clone(),
        network_status: ConnectionStatus::new(),
    };
    let (tx, rx) = watch::channel(initial_payload);


    start_backup_terminal();

    // Task to handle the backup.
    tokio::spawn(async move {
        loop {
            let (socket, _) = listener
                .accept()
                .await
                .expect("Failed to accept backup-connection");
            handle_backup_client(socket, rx.clone()).await;
        }
    });

    // Task for å oppdatere world view
    let tx_clone = tx.clone();
    let wv_rx_clone = wv_watch_rx.clone();

    tokio::spawn(async move {
        loop {
            let new_wv = world_view::get_wv(wv_rx_clone.clone());
            let status = network_watch_rx.borrow().clone();

            let payload = BackupPayload {
                worldview: new_wv,
                network_status: status,
            };

            if tx_clone.send(payload).is_err() {
                println!("Klarte ikkje sende payload til backup-klient");
            }

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

/// Entry point for the backup program (invoked with `cargo run -- backup`).
///
/// Connects to the main process and listens for serialized `BackupPayload`
/// updates over TCP. Displays the current worldview and network status in the terminal.
///
/// # Behavior
/// - Continuously tries to connect to the main process until success or timeout.
/// - Deserializes incoming data and prints system state via `print::worldview`.
/// - If the main process connection fails repeatedly beyond the threshold,
///   the backup promotes itself and returns its local elevator container.
///
/// # Returns
/// - `Some(ElevatorContainer)` if failover is triggered and the backup should take over.
/// - `None` if failover failed or not applicable.
///
/// # Notes
/// In the current solution, this failover logic is disabled using a high timeout.
/// The function is now used solely as a live GUI for displaying the system state.
pub async fn run_as_backup() -> Option<world_view::ElevatorContainer> {
    println!("Starting backup-client...");
    let mut current_wv = init::initialize_worldview(None).await;
    let mut retries = 0;

    loop {
        match timeout(
            config::MASTER_TIMEOUT,
            TcpStream::connect(format!("localhost:{}", config::BCU_PORT))
        ).await {
            Ok(Ok(mut stream)) => {
                retries = 0;
                let mut buf = vec![0u8; 1024];

                loop {
                    match stream.read(&mut buf).await {
                        Ok(0) => {
                            eprintln!("Master connection has ended.");
                            break;
                        },
                        Ok(n) => {
                            let raw = &buf[..n];
                            let payload: Option<BackupPayload> = bincode::deserialize(raw).ok();

                            if let Some(payload) = payload {
                                current_wv = payload.worldview;
                                let status = payload.network_status;

                                print!("\x1B[2J\x1B[H");
                                io::stdout().flush().unwrap();

                                print::worldview(&current_wv, Some(status));
                            } else {
                                println!("Klarte ikkje deserialisere payload.");
                                continue;
                            }
                        },
                        Err(e) => {
                            eprintln!("Error while reading from master: {}", e);
                            break;
                        }
                    }
                }

            },
            _ => {
                retries += 1;
                eprintln!("Failed to connect to master, retry {}.", retries);
                if retries > config::BACKUP_FAILOVER_THRESHOLD {
                    eprintln!("Master failed, promoting backup to master!");
                    // Her kan failover-logikken setjast i gang, t.d. køyre master-logikken.
                    match world_view::extract_self_elevator_container(&current_wv).to_owned() {
                        Some(container) => return Some(container.to_owned()),
                        None => {
                            print::warn(format!("Failed to extract self elevator container"));
                            return None;
                        }
                    }

                }
            }
        }
        sleep(config::BACKUP_RETRY_DELAY).await;
    }
}
```

Innhald frå Rust-filer

Fil: elevator_pro_rebrand\src\config.rs

```rust
//! Keeps some program-parameters, making it easy to change any before building


// TODO: en del av disse blir ikke brukt. Finn ut hvilke, og fjern dem

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 = 50001;
/// 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(5);
/// Period between sending of TCP messages to master-node
pub const TCP_PERIOD: Duration = Duration::from_millis(TCP_PER_U64);
```

Innhald frå Rust-filer

```
/// 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);
/// 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;


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

/// Timeout duration for the master node.
/// This defines how long the backup waits before taking over as master
/// if no connection to the current master is established.
pub const MASTER_TIMEOUT: Duration = Duration::from_millis(50000); // 5 sekunder før failover

/// Timeout duration of backup-nodes
pub const BACKUP_TIMEOUT: Duration = Duration::from_millis(50000); // 5 sekunder før failover

/// How often to send worldview to backup client.
pub const BACKUP_SEND_INTERVAL: Duration = Duration::from_millis(500); // 1 Hz

/// How often to refresh worldview for backup clients.
pub const BACKUP_WORLDVIEW_REFRESH_INTERVAL: Duration = Duration::from_millis(500); // 1 Hz

/// Number of seconds to wait between each retry attempt to connect to master.
pub const BACKUP_RETRY_DELAY: Duration = Duration::from_millis(500);

/// How many failed attempts before we promote backup to master.
pub const BACKUP_FAILOVER_THRESHOLD: u32 = 50;



/// 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\init.rs

```rust
//! ## Initialization Module
//!
//! This module is responsible for initializing the elevator system's worldview
//! and handling system arguments. It provides functions to execute necessary startup tasks.
//!
//! ### Key Responsibilities:
//! - **Worldview Initialization**: Constructs an initial worldview for the elevator system
//!   and attempts to join an existing network if possible.
//! - **Command-line Argument Parsing**: Reads arguments from `cargo run` to control logging
//!   verbosity and enable debug or backup modes.
//! - **Terminal Command Execution**: Provides platform-specific commands for opening new
//!   terminal windows.
//! - **Cost Function Build Execution**: Runs a build script for the hall request assigner
//!   cost function.
//!
//! ### Overview of Functions:
//! - `initialize_worldview`  Creates an initial worldview and merges with the network if possible.
//! - `parse_args`  Parses command-line arguments to configure logging settings and modes.
//! - `get_terminal_command`  Returns the appropriate terminal command for different operating systems.
//! - `build_cost_fn`  Executes a build script for the hall request assigner cost function.

use std::{borrow::Cow, env, net::SocketAddr, 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, 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
/// ```rust
/// let worldview_data: Vec<u8> = initialize_worldview().await;
/// let worldview: worldview::WorldView = worldview::deserialize(&worldview_data);
/// ```
pub async fn initialize_worldview(self_container : Option<&world_view::ElevatorContainer>) -> WorldView {
    let mut worldview = WorldView::default();

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



    // 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 mut wv_from_udp = match check_for_udp().await {
        Some(wv) => wv,
        None => {
            print::info("No other elevators detected on the network.".to_string());
            return worldview
        },
    };

    // 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.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.add_elev(elev_container.clone());

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

```rust
    // Serialize and return the updated worldview
    wv_from_udp
}




/// ### 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>`.
/// 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.");
/// }
/// ```
async fn check_for_udp() -> Option<WorldView> {
    // 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: Option<WorldView>;


    // 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
                read_wv = network::udp_network::parse_message(&buf[..len]);
            }
            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;
            }
        }

        match read_wv {
            Some(wv) => {
                return Some(wv);
            },
            None => {
                continue;
            }
        }
    }

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

    // Return the parsed UDP message data
    None
}


/// ### 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:
/// `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;
        }
    }

    // If no arguments was backup, return 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.
///
```

Innhald frå Rust-filer

```
/// # 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("Failed to run 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;
}

pub async fn build_netimpair_fn() {}
```

Innhald frå Rust-filer

Fil: elevator_pro_rebrand\src\ip_help_functions.rs

```rust
//! This module contains some help functions regarding the IP address
//!
//! Functions
//! - [ip2id]: Generates an ID for the node based on the IP-address.
//! - [get_root_ip]: Extracts the root-ip excluding the ID of the node.

use std::net::IpAddr;
use std::u8;
use crate::config;
use crate::print;



/// Extracts your ID based on `ip`
///
/// ## Example
/// ```
/// use elevatorpro::ip_help_functions::ip2id;
/// use std::net::IpAddr;
/// use std::str::FromStr;
///
/// let ip = IpAddr::from_str("192.168.0.1").unwrap();
/// let id = ip2id(ip);
///
/// assert_eq!(id, 1);
/// ```
///
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('.')      // Split on '.'
        .nth(3)                                    // Extract the 4. element
        .and_then(|s| s.split(':')           // Split on ':' if there was a port
            .next())                               // Only use the part before ':'
        .and_then(|s| s.parse::<u8>().ok());                 // Parse to u8

    match id_str {
        Some(value) => {
            ip_int = value;
        }
        None => {
            print::err(format!("Failed to extract ID from IP"));
        }
    }
    ip_int
}

/// Extracts the root part of an IP address (removes the last segment).
///
/// ## Example
```

```rust
/// ```
/// use std::net::IpAddr;
/// use std::str::FromStr;
/// use elevatorpro::ip_help_functions::get_root_ip;
///
/// let ip = IpAddr::from_str("192.168.0.1").unwrap();
/// let root_ip = get_root_ip(ip);
/// assert_eq!(root_ip, "192.168.0");
/// ```
///
/// 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]; // Remove last element
            root_segments.iter().map(|s| s.to_string()).collect::<Vec<_>>().join(":")
        }
    }
}
```

Innhald frå Rust-filer

Fil: elevator_pro_rebrand\src\lib.rs

```rust
#![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.
//! - **ip_help_functions**: Various helper functions for the local IP-address.
//! - **init**: System initialization.
//! - **manager**: Allocates available tasks to the connected nodes
//! - **network**: Communication between nodes via UDP and TCP, and updateing the worldview locally via mpsc-channels and watch-channels.
//! - **world_view**: The local WorldView
//! - **elevio**: Interface for elevator I/O.
//! - **elevator_logic**: Task execution and reading from the local elevator.
//! - **backup**: Creating, monitoring and running a backup, ready to overtake if the main program crashes

pub mod config;

pub mod ip_help_functions;

pub mod init;

pub mod print;

pub mod manager;

pub mod network;

pub mod world_view;

pub mod elevio;

pub mod elevator_logic;

pub mod backup;
```

Innhald frå Rust-filer

Fil: elevator_pro_rebrand\src\main.rs

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

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() {
    // Check if the program started as backup ("cargo r -- backup")
    let is_backup = init::parse_args();

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

    init::build_cost_fn().await;
    print::info("Starting master process...".to_string());


    /* Initialize a worldview */
    let mut worldview = init::initialize_worldview(self_container.as_ref()).await;
    print::worldview(&worldview, Some(network::ConnectionStatus::new()));


    /* START ----------- Initializing of channels used for the worldview updater ---------------------- */
    let main_mpscs = local_network::Mpscs::new();
    let (wv_watch_tx, wv_watch_rx) = watch::channel(worldview.clone());
    /* END ----------- Initializing of channels used for the worldview updater ---------------------- */

    /* START ----------- Initializing of channels used for the networkstus update ---------------------- */
    let (network_watch_tx, network_watch_rx) = watch::channel(network::ConnectionStatus::new());
    let packetloss_rx = network_watch_rx.clone();
    /* END ----------- Initializing of channels used for the worldview updater ---------------------- */

    // // Send the initialized worldview on the worldview watch, so its not empty when rx tries to borrow it
    let _ = wv_watch_tx.send(worldview.clone());
```

```
/* START ----------- Seperate the mpsc Rx's so they can be sent to the worldview updater ---------------------- */
let mpsc_rxs = main_mpscs.rxs;
/* END ----------- Seperate the mpsc Rx's so they can be sent to the worldview updater ---------------------- */
```

```
/* START ----------- 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;
/* END ----------- Seperate the mpsc Tx's so they can be sent to their designated tasks ---------------------- */
```

```
/* START ----------- Task to watch over the internet connection ---------------------- */
{
    let wv_watch_rx = wv_watch_rx.clone();
    let _network_status_watcher_task = tokio::spawn(async move {
        print::info("Starting to monitor internet".to_string());
        let _ = network::watch_ethernet(wv_watch_rx, network_watch_tx, new_wv_after_offline_tx).await;
    });
}
/* END ----------- Task to watch over the internet connection ---------------------- */
```

```
/* START ----------- Init of channel to send sockets from new TCP-connections on ---------------------- */
let (socket_tx, socket_rx) = mpsc::channel::<(TcpStream, SocketAddr)>(100);
/* START ----------- Init of channel to send sockets from new TCP-connections on ---------------------- */
```

```
/* START ----------- Critical tasks tasks ----------- */
{
    //Continously updates the local worldview
    let _update_wv_task = tokio::spawn(async move {
        print::info("Starting to update worldview".to_string());
        let _ = local_network::update_wv_watch(mpsc_rxs, wv_watch_tx, &mut worldview).await;
    });
}
{
    //Task handling the elevator
    let wv_watch_rx = wv_watch_rx.clone();
    let _local_elev_task = tokio::spawn(async move {
        let _ = elevator_logic::run_local_elevator(wv_watch_rx, elevator_states_tx).await;
    });
}
{
    //Starting the task manager, responsible for delegating tasks
    let wv_watch_rx = wv_watch_rx.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;
    });
}
/* END ----------- Critical tasks tasks ----------- */





/* START ----------- Backup server ----------- */
{
    let wv_watch_rx = wv_watch_rx.clone();
    let _backup_task = tokio::spawn(async move {
        print::info("Starting backup".to_string());
        tokio::spawn(backup::start_backup_server(wv_watch_rx, network_watch_rx));
    });
}
/* END ----------- Backup server ----------- */





/* START ----------- Network related tasks ---------------------- */
{
    //Task listening for UDP broadcasts
    let wv_watch_rx = wv_watch_rx.clone();
    let _listen_task = tokio::spawn(async move {
        print::info("Starting to listen for UDP-broadcast".to_string());
        let _ = udp_network::start_udp_listener(wv_watch_rx, udp_wv_tx).await;
    });
}

{
    //Task sending UDP broadcasts
    let wv_watch_rx = wv_watch_rx.clone();
    let _broadcast_task = tokio::spawn(async move {
        print::info("Starting UDP-broadcaster".to_string());
```

```
            let _ = udp_network::start_udp_broadcaster(wv_watch_rx).await;
        });
    }


    { // UDP EKSPRIMENT
        let wv_watch_rx = wv_watch_rx.clone();
        let _tcp_task = tokio::spawn(async move {
            print::info("Starting UDP direct network".to_string());
            let _ = network::udp_net::start_udp_network(
                wv_watch_rx,
                container_tx,
                packetloss_rx,
                connection_to_master_failed_tx,
                remove_container_tx,
                sent_tcp_container_tx,
            ).await;
        });
    }

    // TCP NED
    // {
    //     //Task handling TCP connections
    //     let wv_watch_rx = wv_watch_rx.clone();
    //     let _tcp_task = tokio::spawn(async move {
    //         print::info("Starting TCP handler".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;
    //     });
    // }

    // {
    //     //Task handling the TCP-listener
    //     let _listener_handle = tokio::spawn(async move {
    //         print::info("Starting tcp listener".to_string());
    //         let _ = tcp_network::listener_task(socket_tx).await;
    //     });
    // }

    // {
    //     //UDP Watchdog
    //     let _udp_watchdog = tokio::spawn(async move {
    //         print::info("Starting udp watchdog".to_string());
    //         let _ = udp_network::udp_watchdog(connection_to_master_failed_tx_clone).await;
    //     });
    // }
    /* START ----------- Network related tasks ---------------------- */


    //Wait before exiting the program
    loop{
        tokio::task::yield_now().await;
    }
}
```

Innhald frå Rust-filer

Fil: elevator_pro_rebrand\src\print.rs

```
//! ## Printing Module
//!
//! This module is onle here to make logging in the terminal easier to read.
//! It allows to print in appropriate colors depening on the situation.
//! It also provides a nice print-format for the WorldView.
use crate::{config, network, world_view::{Dirn, ElevatorBehaviour, WorldView}};
use ansi_term::Colour::{self, Green, Red, Yellow, Purple, White};

use prettytable::color::BLUE;
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: Colour) {
    let print_stat = config::PRINT_ELSE_ON.lock().unwrap().clone();

    if print_stat {
        println!("{}{}\n", color.paint("[CUSTOM]:  "), color.paint(msg));
    }
}

/// 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 {
        println!("{}{}\n", Red.paint("[ERROR]:   "), Red.paint(msg));
    }
}

/// 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 {
        println!("{}{}\n", Yellow.paint("[WARNING]: "), Yellow.paint(msg));
    }
}

/// 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.
```

```rust
///
/// ## 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 {
        println!("{}{}\n", Green.paint("[OK]:      "), Green.paint(msg));
    }
}

/// 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();

    let light_blue = Colour::RGB(102, 178, 255);
    if print_stat {
        println!("{}{}\n", light_blue.paint("[INFO]:    "), light_blue.paint(msg));
    }
}

/// 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();

    let pink = Colour::RGB(255, 51, 255);
    if print_stat {
        println!("{}[MASTER]:  {}\n", pink.paint(""), pink.paint(msg));
    }

}

/// 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();

    let random = Colour::RGB(153, 76, 0);
```

```
    if print_stat {
        println!("{}{}\n", random.paint("[MASTER]:  "), random.paint(msg));
    }
}

/// 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) {
    // Print "[ERROR]:" in red
    print!("{}", Colour::Red.paint("[ERROR]: "));

    // Some colours
    let colors = [
        Colour::Red,
        Colour::Yellow,
        Colour::Green,
        Colour::Cyan,
        Colour::Blue,
        Colour::Purple,
    ];

    // Rest of the print in rainbow
    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()];
        print!("{}", color.paint(c.to_string()));
    }

    println!();
}
/// Pads the input text to a fixed display width using spaces.
```

Innhald frå Rust-filer

```rust
///
/// Accounts for characters that may take more than one column width (e.g. Unicode symbols),
/// ensuring aligned text in terminal-based tables or UI output.
///
/// # Parameters
/// - `text`: The string to pad.
/// - `width`: The total width the text should occupy (including padding).
///
/// # Returns
/// A `String` with the original text left-aligned and padded with spaces to match the desired width.
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))
}

/// Returns a colored and padded string representation of a boolean value.
///
/// Uses green for `true` and red for `false`, and pads the result to a fixed width.
/// Useful for displaying network or status indicators in colored terminal output.
///
/// # Parameters
/// - `value`: The boolean value to represent.
/// - `width`: The width to pad the output to.
///
/// # Returns
/// A colored `String` containing "true" or "false", padded to the given width.
fn colored_bool_label(value: bool, width: usize) -> String {
    let raw_text = if value { "true" } else { "false" };
    let padded = pad_text(raw_text, width); // brukar din hjelpefunksjon
    if value {
        Green.paint(padded).to_string()
    } else {
        Red.paint(padded).to_string()
    }
}

/// Computes an ANSI RGB color escape sequence based on packet loss percentage.
///
/// The color transitions smoothly:
/// - Green at 0% loss (0,255,0)
/// - Yellow at 50% loss (255,255,0)
/// - Red at 100% loss (255,0,0)
///
/// Intended for use in terminal UIs to visually represent loss severity.
///
/// # Parameters
/// - `loss`: Packet loss as a percentage in the range 0100.
///
/// # Returns
/// An ANSI escape string that sets the foreground color for subsequent text.
fn rgb_color_for_loss(loss: u8) -> String {
    // loss frå 0  100 skal gå frå grøn (0,255,0)  gul (255,255,0)  raud (255,0,0)
    let (r, g) = if loss <= 50 {
        let ratio = loss as f32 / 50.0;
        let r = (ratio * 255.0) as u8;
        (r, 255)
    } else {
        let ratio = (loss as f32 - 50.0) / 50.0;
        let g = ((1.0 - ratio) * 255.0) as u8;
        (255, g)
    };
    format!("\x1b[38;2;{};{};0m", r, g)
}

/// Generates a horizontal colored bar representing packet loss visually in the terminal.
///
/// The bar is filled proportionally to the loss percentage, with each filled segment
/// colored using a logarithmic gradient between green and red to emphasize early degradation.
///
/// # Parameters
/// - `loss`: Packet loss as a percentage from 0 to 100.
/// - `width`: The total width (number of characters) of the bar.
///
/// # Returns
/// A `String` containing the ANSI-colored loss bar.
fn colored_loss_bar(loss: u8, width: usize) -> String {
    let mut filled = (loss as usize * width) / 100;
    if loss == 0 {
        filled = 1;
    }

    let mut bar = String::new();

    let k = 20.0; // juster for "kor bratt" det blir i starten

    for i in 0..width {
        let symbol = if i < filled { "" } else { " " };

        let x = i as f32 / width as f32; // 0.0  1.0
        let intensity = ((1.0 + k * x).ln()) / ((1.0 + k).ln()); // logaritmisk interpolering, 01

        let r = (intensity * 255.0) as u8;
        let g = ((1.0 - intensity) * 255.0) as u8;

        let color = format!("\x1b[38;2;{};{};0m", r, g);
        bar.push_str(&format!("{}{}{}", color, symbol, "\x1b[0m"));
    }
    bar
}

/// Logs the current `WorldView` state to the terminal in a structured and colorized table format.
///
/// This function visually presents the status of the elevator system, including:
```

```rust
/// - Network connection status (internet and elevator mesh)
/// - Packet loss as a colored percentage and visual bar
/// - Hall calls across all floors (up/down requests)
/// - Detailed state of each elevator (ID, door, obstruction, last floor, cab requests, hall tasks, etc.)
///
/// The display uses Unicode symbols and ANSI color codes for clarity:
/// - Green/red circles for active/inactive requests
/// - Directional arrows and colors to indicate elevator movement or door states
///
/// # Parameters
/// - `worldview`: A reference to the current global `WorldView` instance.
/// - `connection`: An optional `ConnectionStatus` containing internet and elevator network status
///                 as well as the current packet loss (0100%).
///
/// # Behavior
/// - If configured printing is disabled (`config::PRINT_WV_ON` is false), the function exits early.
/// - Displays high-level metadata, followed by per-elevator breakdown.
/// - Pads and aligns all output to fit well in monospaced terminal environments.
///
/// # Notes
/// - This is intended for human-readable debugging and monitoring purposes.
/// - Printing frequency should be limited (e.g., once per 500 ms).
pub fn worldview(worldview: &WorldView, connection: Option<network::ConnectionStatus> ) {
    let print_stat = config::PRINT_WV_ON.lock().unwrap().clone();
    if !print_stat {
        return;
    }
    // Legg til utskrift av nettverksstatus viss det er med
    println!("{}", ansi_term::Colour::Cyan.bold().paint(""));
    println!("{}", ansi_term::Colour::Cyan.bold().paint("  ELEVATOR NETWORK CONNECTION   "));
    println!("{}", ansi_term::Colour::Cyan.bold().paint(""));
    match connection {
        Some(status) => {
            let on_net_color = colored_bool_label(status.on_internett, 5);
            let elev_net_color = colored_bool_label(status.connected_on_elevator_network, 5);

            let color_prefix = rgb_color_for_loss(status.packet_loss);
            let reset = "\x1b[0m";
            let bar = colored_loss_bar(status.packet_loss, 27);

            println!("");
            println!(" On internett:           {} ", on_net_color);
            println!(" Elevator network:       {} ", elev_net_color);
            println!(" Packet loss:        {}{:>8}%{:>2} ", color_prefix, status.packet_loss, reset);
            println!(" [{}] ", bar);
            println!("");
        }
        None => {
            println!("");
            println!(" Connection status: Not set    ");
            println!("");
        }
```

```
}


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

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

println!(
    " {:<11}  {:<8}                    ",
    worldview.get_num_elev(),
    worldview.master_id
);

for (floor, calls) in worldview.hall_request.iter().enumerate().rev() {
    let up = if floor != worldview.hall_request.len() - 1 {
        if calls[0] { "" } else { "" }
    } else {
        "  " // Ingen opp-knapp i øvste etasje
    };

    let down = if floor != 0 {
        if calls[1] { "" } else { "" }
    } else {
        "  " // Ingen ned-knapp i nederste etasje
    };

    println!(
        " floor:{:<5}           {} {}             ",
        floor,
        down,
        up
    );
}

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 &worldview.elevator_containers {
    let id_text = pad_text(&format!("{}", elev.elevator_id), 4);
                    let door_text = if elev.behaviour == ElevatorBehaviour::DoorOpen || elev.behaviour ==
ElevatorBehaviour::ObstructionError {
    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 num_floors = elev.tasks.len();
let call_list_emoji: Vec<String> = elev.tasks.iter().enumerate().rev()
    .map(|(floor, calls)| {
        let up = if floor != num_floors - 1 {
            if calls[0] { "" } else { "" }
        } else {
            "" // ingen opp-knapp i toppetasjen
        };

        let down = if floor != 0 {
            if calls[1] { "" } else { "" }
        } else {
            "" // ingen ned-knapp i nederste etasje
        };

        format!("{:<2} {} {}", floor, down, up)
    })
    .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::ObstructionError) => pad_text(&Red.paint("Obstruction Error").to_string(), 22),
    (_, ElevatorBehaviour::TravelError) => pad_text(&Red.paint("Travel Error").to_string(), 22),
    (_, ElevatorBehaviour::CosmicError) => pad_text(&Red.paint("Cosmic 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\world_view.rs

```
//! ## WorldView Module
//!
//! This module defines central data structures and helper functions for managing the
//! global state of the elevator network. It contains the `WorldView` struct, which serves as
//! an information packet about the entire system, including all elevators, their tasks, and
//! hall requests. Additionally, it provides serialization, deserialization, and retrieval
//! utilities for handling worldview data efficiently.
//!
//! ### Key Responsibilities:
//! - **Defining Core Structs**: `WorldView` and `ElevatorContainer` store network-wide
//!   and per-elevator state, respectively.
//! - **Handling Directions and States**: The `Dirn` and `ElevatorBehaviour` enums define
//!   movement direction and current operational state of elevators.
//! - **Data Serialization and Deserialization**: Utility functions facilitate efficient
//!   transmission and storage of worldview data.
//! - **Retrieving Elevator Information**: Functions allow querying and modifying elevator
//!   state within the network.
//! - **Master Detection**: Determines whether the current system is the master node.
//!
//! ### Overview of Structs & Enums:
//! - [`Dirn`]  Represents the movement direction of an elevator.
//! - [`ElevatorBehaviour`]  Describes the current state of an elevator.
//! - [`ElevatorContainer`]  Holds information about an individual elevator's state, tasks, and requests.
//! - [`WorldView`]  Contains global network state, including all elevators and hall requests.
//!
//! ### Overview of Functions:
//! - [`serialize`] / [`deserialize`]  Convert worldview data to and from binary format.
//! - [`get_wv`]  Retrieves the latest local worldview.
//! - [`update_wv`]  Updates worldview asynchronously if changes are detected.
//! - [`is_master`]  Checks if the current elevator is the master.
//! - [`extract_elevator_container`] / [`extract_self_elevator_container`]  Retrieve elevator state from worldview.
//! - [`get_index_to_container`]  Finds the index of an elevator container by ID.
//!
//! This module is critical for ensuring a synchronized state across networked elevators.


use serde::{Serialize, Deserialize, de::DeserializeOwned};
use bincode;
use tokio::sync::watch;
use std::collections::HashMap;
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,
    TravelError,
    ObstructionError,
    CosmicError,
}


/// Represents the state of an elevator, including tasks, status indicators, and movement.
#[derive(Serialize, Deserialize, Debug, Clone, PartialEq)]
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,

    /// The last behaviour of the elevator
    pub last_behaviour: ElevatorBehaviour,

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

    /// Indicates wether the stop button is being pressed.
    /// Default: false
    pub stop: 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,
            last_behaviour: ElevatorBehaviour::Idle,
            obstruction: false,
            stop: 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, PartialEq)]
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;
    }
}



/// Serialiserer kva som helst `T` til `Vec<u8>` via bincode
pub fn serialize<T: Serialize>(value: &T) -> Vec<u8> {
    bincode::serialize(value).expect("Klarte ikkje serialisere verdi")
}

/// Deserialiserer `&[u8]` til `T` viss mogleg
pub fn deserialize<T: DeserializeOwned>(buf: &[u8]) -> Option<T> {
    bincode::deserialize(buf).ok()
}


/// 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: &WorldView) -> Option<usize> {
    for i in 0..wv.get_num_elev() {
        if wv.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<WorldView>) -> WorldView {
    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<WorldView>, wv: &mut WorldView) -> 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: &WorldView) -> bool {
    return network::read_self_id() == wv.master_id;
}


/// 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: &WorldView, id: u8) -> Option<&ElevatorContainer> {
    wv.elevator_containers.iter().find(|elevator| elevator.elevator_id == id)
}
```

```
/// 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: &WorldView) -> Option<&ElevatorContainer> {
    let id = network::read_self_id();
    extract_elevator_container(wv, id)
}
```

Innhald frå Rust-filer

Fil: elevator_pro_rebrand\src\elevator_logic\fsm.rs

```
//! Elevator Finite State Machine (FSM) Module
//!
//! This module contains the core logic for the elevator's finite state machine (FSM).
//! It is responsible for reacting to sensor inputs, handling requests, updating the internal
//! elevator state, and controlling motor and lights accordingly.
//!
//! The FSM implements transitions between key elevator states, such as:
//! - Moving
//! - DoorOpen
//! - Idle
//! - Error
//!
//! # Main Responsibilities
//! - Handling initialization from unknown position (`on_init`)
//! - Managing floor arrivals and door timeout logic (`on_floor_arrival`, `on_door_timeout`)
//! - Monitoring inactivity or fault conditions (`handle_error_timeout`)
//! - Executing transitions from Idle state (`handle_idle_state`)
//!
//! # Timers
//! The FSM relies on three coordinated timers:
//! - `door`: Tracks how long the door has been open.
//! - `cab_priority`: Gives passengers time to press cab buttons after door opens.
//! - `error`: Tracks how long the system has been inactive or blocked.
//!
//! These timers are grouped into the [`ElevatorTimers`] struct and passed where needed.
//!
//! # Integration
//! This module is called from the elevator runtime loop and reacts to:
//! - New floor sensor values
//! - Cab and hall call requests
//! - Timer expirations
//!
//!
//! # Related Modules
//! - [`request`]: Direction and behaviour decision logic.
//! - [`self_elevator`]: Local state updates from hardware events.
//! - [`lights`]: Door and button light controls.
//!
//! # Note
//! All function names follow snake_case naming for consistency.


use tokio::time::sleep;
use tokio::sync::mpsc;
use crate::{elevio::{self, elev::Elevator}, 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, request::should_stop, timer::{ElevatorTimers, Timer}};
```

Innhald frå Rust-filer

```
/// Initializes the elevator by moving downward until a valid floor is reached.
///
/// This function sets the elevator in motion and waits until the floor sensor detects
/// a valid floor. During this process, it updates the local elevator state with incoming messages,
/// while tracking timeout conditions through the shared timer structure.
///
/// # Parameters
/// - `self_container`: Mutable reference to the elevator's internal state.
/// - `e`: The hardware-facing elevator handle (for motor control).
/// - `local_elev_rx`: Channel receiver for elevator sensor and button messages.
/// - `timers`: Mutable reference to the shared `ElevatorTimers` instance.
pub async fn on_init(
    self_container: &mut ElevatorContainer,
    e: Elevator,
    local_elev_rx: &mut mpsc::Receiver<elevio::ElevMessage>,
    timers: &mut ElevatorTimers,
) {
    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,
            &mut timers.cab_priority,
            &mut timers.error,
        ).await;

        sleep(config::POLL_PERIOD).await;
    }

    on_floor_arrival(
        self_container,
        e.clone(),
        &mut timers.door,
        &mut timers.cab_priority,
    ).await;
}

/// Handles elevator behavior upon arrival at a new floor.
///
/// If the elevator is currently moving or in an error state, this function checks
/// whether it should stop at the current floor (e.g., due to a hall or cab request).
/// If a stop is needed, it performs the following actions:
/// - Stops the motor
/// - Clears any requests at the current floor
/// - Opens the door and turns on the door light
/// - Starts both the door timer and the cab call priority timer
/// - Sets the elevator's behavior to `DoorOpen`
```

```
///
/// This function is typically called from the main FSM loop or after initialization.
///
/// # Parameters
/// - `elevator`: Mutable reference to the elevator's internal state.
/// - `e`: Elevator hardware interface, used to control motor and lights.
/// - `door_timer`: Timer tracking how long the door should stay open.
/// - `cab_priority_timer`: Timer giving priority to inside cab requests after door opens.
async fn on_floor_arrival(
    elevator: &mut ElevatorContainer,
    e: Elevator,
    door_timer: &mut Timer,
    cab_priority_timer: &mut Timer,
) {
    // Fix startup case: sensor value is 255 when between floors  set to top floor
    if elevator.last_floor_sensor > elevator.num_floors {
        elevator.last_floor_sensor = elevator.num_floors - 1;
    }

    // Turn on light for active cab request (if any)
    lights::set_cab_light(e.clone(), elevator.last_floor_sensor);

    match elevator.behaviour {
        ElevatorBehaviour::Moving | ElevatorBehaviour::ObstructionError | ElevatorBehaviour::TravelError => {
            if request::should_stop(&elevator.clone()) {
                e.motor_direction(Dirn::Stop as u8);
                request::clear_at_current_floor(elevator);
                door_timer.timer_start();
                cab_priority_timer.timer_start();
                elevator.behaviour = ElevatorBehaviour::DoorOpen;
            }
        }
        _ => {}
    }
}

/// Handles the event when the door timeout has expired.
///
/// This function is called after the door has been open for a certain time.
/// It decides what the elevator should do next by:
/// - Choosing a new direction and behaviour using the request logic
/// - Updating the elevator's direction and behaviour accordingly
///
/// If the elevator decides to stay in `DoorOpen`, it means there is still
/// a request at the current floor and the door should remain open.
/// Otherwise, the door light is cleared and the elevator starts moving in the chosen direction.
///
/// # Parameters
/// - `elevator`: Mutable reference to the elevator's internal state.
/// - `e`: Elevator hardware interface, used to control lights and motor.
async fn on_door_timeout(elevator: &mut ElevatorContainer, e: Elevator) {
    match elevator.behaviour {
        ElevatorBehaviour::DoorOpen => {
            let state_pair = request::choose_direction(&elevator.clone());


            elevator.behaviour = state_pair.behaviour;
            elevator.dirn = state_pair.dirn;

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

/// Handles floor arrival updates when a new floor sensor reading is detected.
///
/// If the current floor (`last_floor_sensor`) is different from the previously known floor,
/// this function triggers arrival-handling logic, restarts the error timer,
/// and optionally releases the cab call timer if the stop was due to an inside request.
///
/// # Parameters
/// - `self_container`: Mutable reference to the local elevator state.
/// - `e`: Elevator hardware interface for controlling motor and lights.
/// - `prev_floor`: Mutable reference to the previous floor value (used for change detection).
/// - `timers`: Mutable reference to the shared `ElevatorTimers` instance.
///
/// # Behavior
/// - Calls `on_floor_arrival()` if floor changed.
/// - Starts the error timer on valid floor detection.
/// - Releases the cab call timer if the stop was due to an inside (cab) request.
pub async fn handle_floor_sensor_update(
    self_container: &mut ElevatorContainer,
    e: Elevator,
    prev_floor: &mut u8,
    timers: &mut ElevatorTimers,
) {
    if *prev_floor != self_container.last_floor_sensor {
        on_floor_arrival(self_container, e, &mut timers.door, &mut timers.cab_priority).await;
        timers.error.timer_start();

        // Ignore cab call timeout if request came from inside button
        if !request::was_outside(self_container) {
            timers.cab_priority.release_timer();
        }

        *prev_floor = self_container.last_floor_sensor;
    }
}


pub async fn handle_stop_button(
    self_container: &mut ElevatorContainer,
    e: Elevator,
    prev_stop_btn: &mut bool,
) {
    if *prev_stop_btn != self_container.stop {
        if self_container.stop {
            self_container.behaviour = ElevatorBehaviour::CosmicError;
            e.motor_direction(Dirn::Stop as u8);
        } else {
            self_container.behaviour = ElevatorBehaviour::Idle;
        }
        *prev_stop_btn = self_container.stop;
    }
}

/// Handles door timeout logic when appropriate.
///
/// If the door timer has expired and no obstruction is detected.
/// 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.
///
/// # Behavior
/// - Handles door-close logic via finite state machine if cab call timer is also expired.
pub async fn handle_door_timeout(
    self_container: &mut ElevatorContainer,
    e: Elevator,
    door_timer: &Timer,
    cab_priority_timer: &mut Timer,
) {
    if door_timer.timer_timeouted() && !self_container.obstruction {
        if request::moving_towards_cab_call(&self_container.clone()) {
            cab_priority_timer.release_timer();
        }

        if cab_priority_timer.timer_timeouted() {
            on_door_timeout(self_container, e.clone()).await;
        }
    }
}

/// Monitors elevator activity and triggers error behavior after a timeout period.
```

```rust
///
/// 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_priority_timer`: Timer tracking how long a cab call has been pending.
/// - `error_timer`: Mutable timer for detecting inactivity or system faults.
/// - `prev_cab_priority_timer_stat`: Whether the cab call timer had previously expired.
///
/// # Behavior
/// - Triggers an error state if prolonged inactivity or failure is detected.
pub fn handle_error_timeout(
    self_container: &ElevatorContainer,
    cab_priority_timer: &Timer,
    error_timer: &mut Timer,
    prev_cab_priority_timer_stat: bool,
) {
    if !cab_priority_timer.timer_timeouted() || self_container.behaviour == ElevatorBehaviour::Idle {
        error_timer.timer_start();
    }


    if error_timer.timer_timeouted() && !prev_cab_priority_timer_stat {
        if !self_container.obstruction && (self_container.behaviour == ElevatorBehaviour::DoorOpen) {
            error_timer.timer_start();
        } else {
            print::err("Elevator entered error".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.
pub fn handle_idle_state(
    self_container: &mut ElevatorContainer,
    e: Elevator,
    door_timer: &mut Timer,
) {
    if self_container.behaviour == ElevatorBehaviour::Idle {
        let status_pair = request::choose_direction(&self_container.clone());

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

Innhald frå Rust-filer

Fil: elevator_pro_rebrand\src\elevator_logic\lights.rs

```
//! Elevator Lights Module
//!
//! This module provides utility functions for controlling the elevator's indicator lights.
//!
//! It includes functionality to set:
//! - Hall request lights (`set_hall_lights`)
//! - Cab floor indicator (`set_cab_light`)
//! - Door open light (`set_door_open_light`, `clear_door_open_light`)
//! - Stop button light (`set_stop_button_light`, `clear_stop_button_light`)
//!
//! These lights are updated based on the current elevator state and serialized worldview,
//! and must be explicitly set on each update cycle.
//!
//! # Example
//! ```rust,no_run
//! let e: Elevator = ...;
//! let wv: Vec<u8> = get_serialized_worldview();
//! lights::set_hall_lights(wv, e.clone());
//! lights::set_cab_light(e.clone(), 2);
//! ```
//!
//! # Related
//! See [`world_view`] for worldview structure and serialization logic.

use crate::elevator_logic::ElevatorBehaviour;
use crate::elevio::elev::Elevator;
use crate::world_view::ElevatorContainer;
use crate::world_view::WorldView;


/// 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: &WorldView, e: Elevator, self_container: &ElevatorContainer) {
    for (i, [up, down]) in wv.hall_request.iter().enumerate() {
        let floor = i as u8;
        if floor > e.num_floors {
            break;
        }
```

```
        e.call_button_light(floor, 2, self_container.cab_requests[i]);
        if floor != 0 {
            e.call_button_light(floor, 1, *down);
        }
        if floor != e.num_floors {
            e.call_button_light(floor, 0, *up);
        }
    }

    // Door light
    if self_container.behaviour == ElevatorBehaviour::DoorOpen || self_container.behaviour == ElevatorBehaviour::ObstructionError {
        set_door_open_light(e.clone());
    } else {
        clear_door_open_light(e.clone());
    }
}

/// 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\mod.rs

```rust
//! # Elevator Logic Module
//!
//! This module implements the core logic for the local elevator in the system.
//! It handles:
//! - **Finite State Machine (FSM)** for controlling the motor, doors, and behavior.
//! - **Communication with elevator hardware** to read sensors and activate actuators.
//! - **Integration with the world view**, including request updates and coordination with other elevators.
//! - **Error handling and timing management** to ensure robust operation.

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;
use crate::world_view::WorldView;

/// 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<WorldView>,
    elevator_states_tx: mpsc::Sender<ElevatorContainer>
) {
    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;
        });
    }

    {// lag e til eigen funskjo slik som den over?
        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) {
                    Some(cont) => {
                        lights::set_hall_lights(&wv, e.clone(), &cont);

                    }
                    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.
async fn handle_elevator(
    wv_watch_rx: watch::Receiver<WorldView>,
    elevator_states_tx: mpsc::Sender<ElevatorContainer>,
    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 timers = timer::ElevatorTimers::new(
        Duration::from_secs(3),   // door timer
        Duration::from_secs(10),  // cab call priority
        Duration::from_secs(7),   // error timer
    );


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


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

    loop {
        self_elevator::update_elev_container_from_msgs(
            &mut local_elev_rx,
            &mut self_container,
            &mut timers.cab_priority ,
            &mut timers.error
        ).await;
        /*==========================================================================*/
        /*                    START: FSM Events                    */
        /*==========================================================================*/
        fsm::handle_idle_state(
            &mut self_container,
            e.clone(),
            &mut timers.door
        );
        fsm::handle_floor_sensor_update(
            &mut self_container,
            e.clone(),
            &mut prev_floor,
            &mut timers,
        ).await;
        fsm::handle_door_timeout(
            &mut self_container,
            e.clone(),
            &timers.door,
            &mut timers.cab_priority,
        ).await;
        fsm::handle_stop_button(
            &mut self_container,
            e.clone(),
            &mut prev_stop_btn
        ).await;
        fsm::handle_error_timeout(
            &self_container,
            &timers.cab_priority,
            &mut timers.error,
            timers.prev_cab_priority_timeout,
        );
        /*==========================================================================*/
        /*                    END: FSM Events                    */
        /*==========================================================================*/


/*==========================================================================================
================================================*/
        sleep(config::POLL_PERIOD).await;

        update_motor_direction_if_needed(&self_container, &e);
        update_error_state(&mut self_container, &timers.error, &mut timers.prev_cab_priority_timeout, &prev_behavior);
        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);
        self_container.last_behaviour = last_behavior;

        //Hent nyeste worldview
```

```
        if world_view::update_wv(wv_watch_rx.clone(), &mut wv).await{
            update_tasks_and_hall_requests(&mut self_container, &wv).await;
        }
        //Send til update_wv -> nye self_container
        let _ = elevator_states_tx.send(self_container.clone()).await;
        yield_now().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: &WorldView
){
    if let Some(task_container) = world_view::extract_self_elevator_container(wv) {
        self_container.tasks = task_container.tasks.clone();
        self_container.cab_requests = task_container.cab_requests.clone();
        self_container.unsent_hall_request = task_container.unsent_hall_request.clone();
    } 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<WorldView>
) -> ElevatorContainer {
    loop {
        let wv = world_view::get_wv(wv_rx.clone());
        if let Some(container) = world_view::extract_self_elevator_container(&wv) {
            return container.clone();
        } else {
            print::warn(format!("Failed to extract self elevator container, retrying..."));
            sleep(Duration::from_millis(100)).await;
        }
    }
}


/// 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.
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);
    }
}

/// 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_priority_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_priority_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.
fn update_error_state(
    self_container: &mut ElevatorContainer,
    error_timer: &timer::Timer,
    prev_cab_priority_timer_stat: &mut bool,
    prev_behavior: &ElevatorBehaviour,
) {
    if error_timer.timer_timeouted() {
        if was_prew_state_error(prev_behavior){ return;}
        *prev_cab_priority_timer_stat = true;
        if *prev_behavior == ElevatorBehaviour::DoorOpen {
            self_container.behaviour = ElevatorBehaviour::ObstructionError;

        } else if *prev_behavior == ElevatorBehaviour::Moving {
            self_container.behaviour = ElevatorBehaviour::TravelError;
        } else {
            self_container.behaviour = ElevatorBehaviour::CosmicError;
        }
    } else {
        *prev_cab_priority_timer_stat = false;
    }
}

fn was_prew_state_error(prev_behavior:  &ElevatorBehaviour) -> bool{
    *prev_behavior == ElevatorBehaviour::ObstructionError ||
    *prev_behavior == ElevatorBehaviour::TravelError ||
    *prev_behavior == ElevatorBehaviour::CosmicError
}

/// 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.
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`.
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::ObstructionError {
        self_container.dirn = Dirn::Stop;
    }
}
```

Innhald frå Rust-filer

Fil: elevator_pro_rebrand\src\elevator_logic\request.rs

```rust
//! Elevator request evaluation and direction decision logic.
//!
//! This module provides helper functions for determining the next action of an elevator,
//! based on its current direction, pending requests, and behavioural state.
//!
//! It is used as part of the elevator finite state machine (FSM) and ensures deterministic,
//! stateless logic for evaluating what the elevator should do at any given point in time.
//!
//! # Overview
//! The core functionality includes:
//! - Checking for cab or hall requests above or below the current floor.
//! - Determining whether to stop at the current floor.
//! - Choosing direction and behaviour based on task layout.
//! - Inferring whether the elevator is moving towards a cab request.
//!
//! # Primary Structs
//! - [`DirnBehaviourPair`]: Return value combining direction and behaviour (e.g., Moving Up).
//!
//! # Behaviour
//! The logic is stateless and purely functional, based on snapshot data of the elevator's state.
//! Each function expects a reference to the full elevator state (`ElevatorContainer`),
//! and returns either a boolean, a direction, or a `DirnBehaviourPair`.
//!
//! This ensures consistency, testability, and reusability of logic across different parts of the system.
//!
//! # Example
//! ```rust,no_run
//! use elevator_logic::request::{choose_direction, should_stop};
//! use world_view::{ElevatorContainer, Dirn};
//!
//! let direction_and_behaviour = choose_direction(&elevator);
//! if should_stop(&elevator) {
//!     // Open doors, reset timers
//! }
//! ```

use crate::world_view::{Dirn, ElevatorBehaviour, ElevatorContainer};

/// Represents a combination of a direction and an elevator behaviour state.
///
/// Typically used as the return type for direction decision functions,
/// such as in the elevator finite state machine.
#[derive(Debug, Clone, Copy)]
pub struct DirnBehaviourPair {
    /// direction of the elevator
    pub dirn: Dirn,

    /// the behavior of the elevator
    pub behaviour: ElevatorBehaviour,
}
```

```
/// Checks if there are any hall or cab requests above the elevator's current floor.
///
/// Returns `true` if any requests exist on floors higher than the current one, otherwise `false`.
///
/// # Parameters
/// - `elevator`: Reference to the elevator's internal state.
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
}

/// Checks if there are any **cab requests** (inside elevator) above the current floor.
///
/// Returns `true` if any cab calls exist above, otherwise `false`.
///
/// # Parameters
/// - `elevator`: Reference to the elevator's internal state.
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
}

/// Checks if there are any hall or cab requests below the elevator's current floor.
///
/// Returns `true` if any requests exist on floors lower than the current one, otherwise `false`.
///
/// # Parameters
/// - `elevator`: Reference to the elevator's internal state.
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
}


/// Checks if there are any **cab requests** (inside elevator) below the current floor.
///
/// Returns `true` if any cab calls exist below, otherwise `false`.
///
/// # Parameters
/// - `elevator`: Reference to the elevator's internal state.
fn insie_below(elevator: &ElevatorContainer) -> bool {
    for floor in 0..elevator.last_floor_sensor as usize {
        if elevator.cab_requests[floor] {
            return true;
        }
    }
    false
}

/// Checks for any pending tasks or cab requests at the elevator's current floor.
///
/// Returns `true` if there is a request at the current floor, otherwise `false`.
///
/// # Parameters
/// - `elevator`: Reference to the elevator's internal state.
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
}

/// Determines the intended direction of travel based on current tasks at the elevator's floor.
///
/// Returns:
/// - `Dirn::Up` if there's an up request
/// - `Dirn::Down` if there's a down request
/// - `Dirn::Stop` if no direction is requested
///
/// # Parameters
/// - `elevator`: Reference to the elevator's internal state.
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;
    }

}

/// Determines whether the elevator is moving towards any cab request (inside call).
///
/// This is used to decide whether to stop even if there are no hall requests.
///
/// # Parameters
/// - `elevator`: Reference to the elevator's internal state.
///
/// # Returns
/// `true` if there is a cab call in the current direction of travel, otherwise `false`.
pub fn moving_towards_cab_call(elevator: &ElevatorContainer) -> bool {
    if elevator.last_floor_sensor == elevator.num_floors-1 || elevator.last_floor_sensor == 0 {
        return true;
    }
    match elevator.dirn {
        Dirn::Up => {
            return inside_above(&elevator.clone());
        },
        Dirn::Down => {
            return insie_below(&elevator.clone());
        },
        Dirn::Stop => {
            return false;
        }
    }
}

/// Main decision logic to determine the elevator's next direction and behaviour.
///
/// Uses the elevator's current direction, position, and requests above/below to return a
/// `DirnBehaviourPair` representing whether to move, open the door, or stay idle.
///
/// # Parameters
/// - `elevator`: Reference to the elevator's internal state.
///
/// # Returns
/// A `DirnBehaviourPair` representing the chosen direction and behaviour state.
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 {
                DirnBehaviourPair { dirn: Dirn::Stop, behaviour: ElevatorBehaviour::Idle }
            }
        }
    }
}

/// Determines whether the elevator should stop at the current floor.
///
/// This decision depends on cab calls and hall calls at the floor,
/// and whether there are pending requests in the current direction.
///
/// # Parameters
/// - `elevator`: Reference to the elevator's internal state.
///
/// # Returns
/// `true` if the elevator should stop, otherwise `false`.
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,
    }
}


/// Evaluates whether the elevator was previously outside its designated service range.
///
/// This function mirrors `should_stop()` logic and may be used for debugging or fallback decisions.
///
/// # Parameters
/// - `elevator`: Reference to the elevator's internal state.
///
/// # Returns
/// `true` if considered outside or should stop, otherwise `false`.
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,
    }
}

/// Clears any active cab request at the elevator's current floor.
///
/// This function assumes the elevator has stopped at the correct floor
/// and has fulfilled the passenger's request.
///
/// # Parameters
/// - `elevator`: Mutable reference to the elevator's internal state.
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;
        },
    }
}
```

Fil: elevator_pro_rebrand\src\elevator_logic\self_elevator.rs

```
//! # Local Elevator Module
//!
//! This module is responsible for managing the local elevator instance, including:
//! - Initializing the local elevator (`init`)
//! - Handling communication with the elevator server (`start_elevator_server`)
//! - Polling and processing elevator sensor data (`read_from_local_elevator`)
//! - Updating the local elevator state (`update_elev_container_from_msgs`)
//!
//! ## Overview
//! The module establishes a communication channel with the local elevator hardware, allowing
//! sensor readings (call buttons, floor sensors, stop button, obstruction status) to be processed
//! asynchronously. It also provides an interface for starting the elevator server on different platforms.
//!
//!
//! ## Functionality
//! - **Initialization**: Sets up the elevator instance, starts the elevator server, and initializes
//!   message polling from the hardware.
//! - **Message Handling**: Processes messages received from the elevator, updating the `ElevatorContainer`
//!   accordingly.
//! - **Asynchronous Processing**: Uses `tokio` tasks to handle sensor polling and inter-process communication.

use tokio::time::{sleep, Duration};
use crossbeam_channel as cbc;
use tokio::process::Command;
use tokio::sync::mpsc;

use crate::network;
use crate::world_view::ElevatorContainer;
use crate::{world_view::ElevatorBehaviour, 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 }
        }
    }
}



/// ### Get local IP address
fn get_ip_address() -> String {
    let self_id = network::read_self_id();
    format!("{}.{}", config::NETWORK_PREFIX, self_id)
}

/// ### Starts the elevator_server
async fn start_elevator_server() {
    let ip_address = get_ip_address();
    let ssh_password = "Sanntid15"; // Hardkodet passord, vurder sikkerhetsrisiko

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

        // Start the elevator server without opening a terminal
        let elevator_server_command = format!(
            "sshpass -p '{}' ssh student@{} 'nohup elevatorserver > /dev/null 2>&1 &'",
            ssh_password, ip_address
        );

        print::info(format!("\nStarting elevatorserver in new terminal:\n\t{}", elevator_server_command));

        let _ = Command::new("sh")
            .arg("-c")
            .arg(&elevator_server_command)
            .output().await
            .expect("Error while starting elevatorserver");
    }

    print::ok(format!("Elevator server started."));
}

// ### 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("Error while initiating elevator");

    // 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
}

/// Send forth messages from local elevator to worldview updater
async fn read_from_local_elevator(rxs: LocalElevRxs, local_elev_tx: mpsc::Sender<elevio::ElevMessage>) -> std::io::Result<()> {
    loop {
        if let Ok(call_button) = rxs.call_button.try_recv() {
            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() {
            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() {
            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() {
            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;
        }
        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_priority_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_priority_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));
                        if let Some(stop) = msg.stop_button {
                            container.stop = stop;
                        }
                    }

                    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\timer.rs

```
//! Timer module for managing asynchronous timeouts in elevator control logic.
//!
//! This module defines two core components:
//! - [`Timer`]: A general-purpose timer that supports both soft (elapsed time) and hard (manual) timeouts.
//! - [`ElevatorTimers`]: A struct that bundles all timers used in the elevator FSM, including door timeout,
//!   cab call priority grace period, and general error detection.
//!
//! These components are used throughout the elevator state machine to control behavior based on timeouts,
//! such as how long to keep doors open, how long to prioritize internal cab calls, or when to enter an error state
//! due to inactivity or unresponsiveness.
//!
//! # Usage
//! Timers are used in `fsm.rs` to manage:
//! - Door open duration (`door` timer)
//! - Grace period for cab button prioritization (`cab_priority` timer)
//! - Communication or logic errors (`error` timer)
//!
//! # Example (using Timer standalone)
//! ```rust,no_run
//! use tokio::time::Duration;
//! use crate::elevator_logic::timer::Timer;
//!
//! let mut door_timer = Timer::new(Duration::from_secs(3));
//! door_timer.timer_start();
//!
//! // In FSM loop:
//! if door_timer.timer_timeouted() {
//!     // Trigger door close logic
//! }
//! ```
//!
//! # ElevatorTimers Usage
//! ElevatorTimers simplifies timer management by grouping all related timers into one struct:
//!
//! ```rust,no_run
//! let mut timers = ElevatorTimers::new(
//!     Duration::from_secs(3),   // door
//!     Duration::from_secs(10),  // cab priority
//!     Duration::from_secs(7),   // error
//! );
//!
//! timers.door.timer_start();
//! if timers.cab_priority.timer_timeouted() {
//!     // Prioritization window over
//! }
//! ```
//!
//! # Timer Behavior
//! - A timer is **inactive** until [`timer_start()`](Timer::timer_start) is called.
//! - Once active, it compares current time with the internal start time.
```

```
//! - The timer can be **manually forced** to timeout using [`release_timer()`](Timer::release_timer).
//! - A call to [`timer_timeouted()`](Timer::timer_timeouted) returns `true` if either a soft or hard timeout has occurred.
//!
//! # Related
//! Used heavily in the [`fsm`](crate::elevator_logic::fsm) module.


use tokio::time::Duration;

/// A simple timer utility for managing soft and hard timeouts in asynchronous contexts.
///
/// The timer can be started and queried to check whether the timeout duration has been exceeded.
/// In addition to the regular (soft) timeout based on elapsed time, a "hard timeout" flag can be manually triggered
/// to force the timer into a timeout state regardless of elapsed time.
pub struct Timer {
    hard_timeout: bool,
    timer_active: bool,
    timeout_duration: tokio::time::Duration,
    start_time: tokio::time::Instant,
}

impl Timer {
    /// Creates and returns a new timer instance.
    ///
    /// The timer is initially inactive and has not timed out.
    ///
    /// # Arguments
    /// * `timeout_duration`  The duration after which the timer should timeout once started.
    ///
    /// # Returns
    /// A new `Timer` instance with the specified timeout duration.
    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(),
        }
    }
    /// Starts the timer by setting it as active and resetting the start time.
    ///
    /// This also clears any manually set hard timeout.
    pub fn timer_start(&mut self) {
        self.hard_timeout = false;
        self.timer_active = true;
        self.start_time = tokio::time::Instant::now();
    }

    /// Forces the timer into a timeout state, regardless of elapsed time.
    ///
    /// This is useful for emergency shutdowns or forced exits.
    pub fn release_timer(&mut self) {
        self.hard_timeout = true;
    }

    /// Returns the duration elapsed since the timer was last started.
    ///
    /// This does not check whether the timer is active or has timed out.
    pub fn get_wall_time(&mut self) -> tokio::time::Duration {
        return tokio::time::Instant::now() - self.start_time
    }



    /// Checks if the timer has timed out.
    ///
    /// The timer is considered timed out if:
    /// - It is active and the elapsed time exceeds `timeout_duration`, or
    /// - It has been manually forced to timeout using `release_timer()`.
    ///
    /// # Returns
    /// `true` if the timer is considered to have timed out; `false` otherwise.
    pub fn timer_timeouted(&self) -> bool {
        return (self.timer_active && (tokio::time::Instant::now() - self.start_time) > self.timeout_duration) || self.hard_timeout;
    }
}


/// Collection of timers used in the elevator's finite state machine (FSM).
///
/// This struct encapsulates all timers that track different timeout conditions
/// such as door closing, inside call priority window, and general error state.
/// Also includes state tracking related to inside call grace period.
pub struct ElevatorTimers {
    /// Timer for automatic door closing.
    pub door: Timer,

    /// Timer that provides a short grace period to prioritize inside (cab) calls
    /// after a passenger enters the elevator.
    ///
    /// When the elevator stops at a floor due to a hall request (e.g. someone pressed "up"),
    /// this timer is started to give the passenger a few seconds to press a cab button
    /// (e.g. "Floor 3"). During this grace period, the FSM prioritizes inside orders
    /// in the direction the elevator was called.
    ///
    /// After the timer expires, the elevator becomes available for other external requests.
    pub cab_priority: Timer,

    /// Timer for tracking long-term inactivity or error conditions.
    pub error: Timer,

    /// Tracks whether the `cab_priority` timer had timed out in the previous iteration.
    pub prev_cab_priority_timeout: bool,
}
```

Innhald frå Rust-filer

```
impl ElevatorTimers {
    /// Creates a new `ElevatorTimers` instance with custom durations.
    ///
    /// # Parameters
    /// - `door_duration`: Duration before automatically closing the elevator door.
    /// - `cab_priority_duration`: Grace period for prioritizing cab calls after stopping.
    /// - `error_duration`: Duration before considering the elevator to be in an error state.
    ///
    /// # Returns
    /// An initialized `ElevatorTimers` struct with the specified timeout settings.
    pub fn new(
        door_duration: Duration,
        cab_priority_duration: Duration,
        error_duration: Duration,
    ) -> Self {
        ElevatorTimers {
            door: Timer::new(door_duration),
            cab_priority: Timer::new(cab_priority_duration),
            error: Timer::new(error_duration),
            prev_cab_priority_timeout: false,
        }
    }
}
```

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

```
//! ## Elevator I/O module for the local elevator
//!
//! This module is mostly consisting of handed out resources, but some functionality were added.
//! The handed out functionality is placed in the submodules [`elev`] and [`poll`].
//!
//! Additional functionality includes message handling for elevator events,
//! call button state management, and conversion utilities for call types.
//!
//! ## Overview
//! This module provides data structures and utilities for handling elevator
//! input/output operations. It includes:
//!
//! - `ElevMsgType`: Enum representing different elevator events.
//! - `ElevMessage`: Struct for wrapping elevator messages.
//! - `CallType`: Enum for representing call button types.
//! - `CallButton`: Struct for representing call button presses, including
//!    floor, call type, and elevator ID.
//!
//! These components allow structured handling of elevator input events, ensuring
//! that different types of messages (such as button presses and sensor activations)
//! are processed in a uniform manner.

#[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);
        }
    }
}
```

Fil: elevator_pro_rebrand\src\elevio\poll.rs

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

use crossbeam_channel as cbc;
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\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::{config, world_view::{ElevatorBehaviour, WorldView}};

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

/// This function executes the cost algorithm, and returns the output
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()
}

/// This function creates the input to the cost function algorithm based on the worldview
pub async fn create_hall_request_json(wv: &WorldView) -> Option<String> {
    let mut states = HashMap::new();
    for elev in wv.elevator_containers.iter() {
        let key = elev.elevator_id.to_string();
          if elev.behaviour != ElevatorBehaviour::TravelError && elev.behaviour != ElevatorBehaviour::ObstructionError &&
elev.behaviour != ElevatorBehaviour::CosmicError {
            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 {
                    config::ERROR_ID as i32
                },
                direction: format!("{:?}", elev.dirn.clone()).to_lowercase(),
                cabRequests: elev.cab_requests.clone(),
                },
            );
        }
    }

    if states.is_empty() {
        return None
    }
    let request = HallRequests {
        hallRequests: wv.hall_request.clone(),
        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)
}
```

Innhald frå Rust-filer

Fil: elevator_pro_rebrand\src\manager\mod.rs

```rust
//! ## Manager Module
//!
//! This module is responsible for allocating tasks on the network.
//! It executes the hall assigner script give under project resources.

use std::collections::HashMap;
use tokio::{sync::{mpsc, watch}, time::sleep};
use crate::{config, world_view::{self, WorldView}};
use crate::print;
mod json_serial;



/// Main task for managing elevator coordination.
///
/// Continuously listens for updates in the global world view (`wv_watch_rx`).
/// If the current node is the designated master, it computes and distributes tasks
/// to all known elevators using a cost-based assignment algorithm.
/// If the node is not the master, it waits for a defined slave timeout before checking again.
///
/// Behavior:
/// - Master nodes actively calculate and delegate hall requests.
/// - Slave nodes remain idle and periodically check for changes in master status.
///
/// Parameters:
/// - `wv_watch_rx`: A watch channel providing updates to the shared world view state.
/// - `delegated_tasks_tx`: A channel used to send the delegated hall tasks to other modules.
pub async fn start_manager(
    wv_watch_rx: watch::Receiver<WorldView>,
    delegated_tasks_tx: mpsc::Sender<HashMap<u8, Vec<[bool; 2]>>>
) {
    let mut wv = world_view::get_wv(wv_watch_rx.clone());

    loop {
        // Update local copy of the world view
        if world_view::update_wv(wv_watch_rx.clone(), &mut wv).await {
            // Check if this node is the master
            if world_view::is_master(&wv) {
                // Calculate and send out delegated hall requests
                let _ = delegated_tasks_tx.send(get_elev_tasks(&wv).await).await;
            } else {
                // If not master, wait before checking again
                sleep(config::SLAVE_TIMEOUT).await;
            }
        }

        // Polling delay to limit update frequency
        sleep(config::POLL_PERIOD).await;
    }
}
```

```rust
/// Generates a set of hall requests assigned to each elevator based on cost minimization.
///
/// This function first serializes the global world view to JSON,
/// then passes the data to an external cost algorithm module which calculates the most optimal
/// assignment of hall calls to elevators. If successful, it returns a map of elevator IDs
/// to their respective assigned requests.
///
/// Behavior:
/// - If the cost algorithm returns a valid JSON string, it is parsed and returned as a HashMap.
/// - If the cost algorithm fails or returns an empty string, an error is logged and an empty map is returned.
/// - If JSON parsing fails, an error is also logged and an empty map is returned.
///
/// Parameters:
/// - `wv`: The serialized global world view as a byte vector.
///
/// Returns:
/// - A `HashMap` where each key is an elevator ID (`u8`), and each value is a list of `[bool; 2]`
///   arrays indicating hall call assignments (up/down).
async fn get_elev_tasks(wv: &WorldView) -> 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() {
            print::err(format!(
                "run_cost_algorithm returned an empty string"
            ));
            return HashMap::new();
        }

        return serde_json::from_str(&json_cost_str).unwrap_or_else(|e| {
            print::err(format!("Failed to parse JSON from cost algorithm: {}", e));
            HashMap::new()
        });
    }

    print::err("create_hall_request_json returned None.".to_string());
    HashMap::new()
}
```

Innhald frå Rust-filer

Fil: elevator_pro_rebrand\src\network\mod.rs

```
//! ## Network module
//!
//! This module is responsible for the network in the system, and is in some sense the most sentral part in the whole project.
//! The module has a lot of responsibilities, and is therefore splittet into a few sub-modules.
//!
//! ## Sub-modules
//! - [tcp_network]
//! - [udp_network]
//! - [local_network]
//!
//! ## Key Features
//! - Using UDP broadcast to publish WorldView on the network, and detecting a network when starting up.
//! - Using TCP to share elevator-spesific data from slave-nodes to master-nodes.
//! - Using a set of thread-safe channels to let different parts of the program to share information.
//! - Monitoring the network, automatically detecting connection loss and unoperatable levels of packetloss
//!
//! ## Functions
//! - `watch_ethernet`: Updates the network status, making sure the program detects connection loss and high packet loss
//! - `read_network_status`: Gives a boolean indicating if your network connection is operatable.

pub mod tcp_network;
pub mod udp_network;
pub mod local_network;
pub mod udp_net;


use crate::world_view::WorldView;
use crate::{init, config, print, ip_help_functions, world_view, };
use serde::{Serialize, Deserialize};
use tokio::net::UdpSocket;
use tokio::time::{timeout, Duration, Instant};
use tokio::sync::{mpsc, watch};
use std::sync::atomic::{Ordering, AtomicU8, AtomicBool};
use std::sync::OnceLock;
use std::thread::sleep;
use local_ip_address::local_ip;
use std::net::IpAddr;

#[derive(Serialize, Deserialize, Clone, Debug)]
pub struct ConnectionStatus {
    /// true if we have network on the subnet, regardless of the packet loss
    pub on_internett: bool,

    /// true if we decide to be connected to the elevator network
    pub connected_on_elevator_network: bool,

    /// percentage of packet loss (0 - 100)%
    pub packet_loss: u8,
}

impl ConnectionStatus {
    /// Lag ein ny standard status (alt false / 0%)
    pub fn new() -> Self {
        Self {
            on_internett: false,
            connected_on_elevator_network: false,
            packet_loss: 0,
        }
    }
    ///convert float to a percentage
    pub fn set_packet_loss(&mut self, loss: f32) {
        self.packet_loss = (loss * 100.0) as u8;
    }
}

/// 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
/// }
/// ```
fn get_self_ip() -> Result<IpAddr, local_ip_address::Error> {
    let ip = match local_ip() {
        Ok(ip) => {
            ip
        }
        Err(e) => {
            return Err(e);
        }
    };
    Ok(ip)
}

// Må oppdateres
// / 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<WorldView>,
    network_watch_tx: watch::Sender<ConnectionStatus>,
    new_wv_after_offline_tx: mpsc::Sender<WorldView>
) {
    let mut last_net_status = false;
    // TODO:: legge på hystesrese
    let network_quality_rx = start_packet_loss_monitor(
        1,
        5,
        1000 as usize,
        1.0
    ).await;


    loop {
        let ip = get_self_ip();
        let mut connection_status = ConnectionStatus::new();
        let mut net_status = false;

        match ip {
            Ok(ip) if ip_help_functions::get_root_ip(ip) == config::NETWORK_PREFIX => {
                let (is_ok, loss)  = network_quality_rx.borrow().clone();
                net_status = is_ok;

                connection_status.on_internett = true;
                connection_status.connected_on_elevator_network = is_ok;
                connection_status.set_packet_loss(loss);

                let _ = network_watch_tx.send(connection_status.clone());
            }
            _ => {
                // Mistet IP eller feil subnet  nullstill status
                connection_status.on_internett = false;
```

```
                connection_status.connected_on_elevator_network = false;
                connection_status.packet_loss = 100;
                net_status = false;

                let _ = network_watch_tx.send(connection_status.clone());
            }
        }

        if last_net_status != net_status {
            if net_status {
                // Gjekk frå offline  online
                let mut wv = world_view::get_wv(wv_watch_rx.clone());
                let self_elev = world_view::extract_self_elevator_container(&wv);
                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;
        }

        sleep(config::POLL_PERIOD);
    }
}


async fn wait_for_ip() -> IpAddr {
    loop {
        if let Ok(ip) = get_self_ip() {
            return ip;
        } else {
            sleep(config::POLL_PERIOD);
        }
    }
}

/// Startar ein pingmålar som returnerer (status, pakketap)
///
/// - `interval_ms`: tid mellom ping
/// - `timeout_ms`: ping-timeout
/// - `max_window`: storleik på glidande vindu
/// - `max_loss_rate`: maks akseptert pakketap (t.d. 0.2 for 20%)
///
/// Retur: `watch::Receiver<(bool, f32)>`, der:
/// - `true` betyr OK (god nok kvalitet)
/// - `f32` er pakketap (0.01.0)
pub async fn start_packet_loss_monitor(
    interval_ms: u64,
    timeout_ms: u64,
    max_window: usize,
    max_loss_rate: f32,
) -> watch::Receiver<(bool, f32)> {
    use tokio::sync::watch;
    use socket2::{Socket, Domain, Type};
    let (tx, rx) = watch::channel((true, 0.0)); // start som OK
    let addr = format!("{}:{}", wait_for_ip().await, config::DUMMY_PORT);


    tokio::spawn(async move {
        let mut last_loss: f32 = 0.0;
        let mut last_status: bool = false;
        let mut last_instant = Instant::now();
        let mut window: VecDeque<bool> = VecDeque::from(vec![true; max_window]);

        loop {
            // Send ping
            let success = {
                let socket_addr: std::net::SocketAddr = match addr.parse() {
                    Ok(addr) => addr,
                    Err(_) => {
                        break false;
                    }
                };

                // Opprett socket
                let socket_temp = match Socket::new(Domain::IPV4, Type::DGRAM, None) {
                    Ok(s) => s,
                    Err(_) => {
                        break false;
                    }
                };

                if socket_temp.set_nonblocking(true).is_err() {break false}

                if socket_temp.set_reuse_address(true).is_err() {break false}

                if socket_temp.set_broadcast(true).is_err() {break false}

                if socket_temp.bind(&socket_addr.into()).is_err() {break false}

                match UdpSocket::from_std(socket_temp.into()) {

                    Ok(socket) => {
                        let payload = b"ping";
                        if socket.send_to(payload, &addr).await.is_err() {
                            false
                        } else {
                            let mut buf = [0u8; 16];
                            timeout(Duration::from_millis(timeout_ms), socket.recv_from(&mut buf))
                            .await
                            .ok()
                            .map(|r| r.is_ok())
                            .unwrap_or(false)
                        }
                    }
                    Err(_) => {
                        false
                    },
                }

            };


            // Oppdater vindu
            window.push_back(success);
            if window.len() > max_window {
                window.pop_front();
            }

            // Berekn tap i vinduet
            let fail_count = window.iter().filter(|&&ok| !ok).count();
            let raw_loss = fail_count as f32 / window.len() as f32;
            let loss_rate = 1.0 - (1.0 - raw_loss).sqrt();


            let new_status = loss_rate <= max_loss_rate;
            // Send ny status viss han har endra seg
            if (last_status != new_status) || (loss_rate - last_loss).abs() > 0.01 {
                if Instant::now() - last_instant > Duration::from_secs(5) {
                    last_instant = Instant::now();
                    let _ = tx.send((new_status, loss_rate));
                    last_status = new_status;
                }else {
                    let _ = tx.send((new_status, loss_rate));
                    last_loss = loss_rate;
                }
            }

            // Pause før neste ping
            tokio::time::sleep(Duration::from_millis(interval_ms)).await;
        }
    });

    rx
}



use std::collections::VecDeque;
fn moving_average(samples: &VecDeque<u64>) -> f64 {

    if samples.is_empty() {
        return f64::INFINITY;
    }
    let sum: u64 = samples.iter().sum();
    sum as f64 / samples.len() as f64
}

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);
}
```

Fil: elevator_pro_rebrand\src\network\tcp_network.rs

```
//! # TCP Module
//!
//! This module handles TCP communication between master and slave systems in a network.
//! It includes functions for setting up listeners, managing connections, and transferring data.
//!
//! ## Functions
//! - `listener_task`: Listens for incoming TCP connections.
//! - `tcp_handler`: Manages TCP communication by switching between master and slave behavior.
//!
//! ## Key Features
//! - Manages the communication between nodes, dynamically changing behaviour when a node switches from master
&harr; slave.
//! - All nodes has an active listener, accepting incoming connections.
//! - Slave nodes sends [ElevatorContainer]'s to the master.
//! - Master nodes recieves [ElevatorContainer]'s from slaves.
//! - All connections are set up on configured sockets, to handle extreme (>50%) packetloss.
//!
//! ## Usage
//! The module integrates with the system's network and worldview components to facilitate
//! reliable master-slave communication over TCP.

use std::{io::Error, sync::atomic::{AtomicBool, Ordering}};
use tokio::{io::{AsyncReadExt, AsyncWriteExt}, net::{TcpListener, TcpStream}, sync::{mpsc, watch}, task::JoinHandle,
time::{sleep, Duration}};
use std::net::SocketAddr;
use socket2::{Domain, Protocol, SockAddr, Socket, TcpKeepalive, Type};
use crate::{config, ip_help_functions::{self}, network, print, world_view::{self, ElevatorContainer, WorldView}};


/* __________ 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 stream and address for 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 socket = match create_tcp_socket() {
        Ok(sock) => sock,
        Err(e) => {
            print::err(format!("Failed to set up TCP listener: {}", e));
            panic!();
        }
    };

    let listener = match listener_from_socket(socket) {
        Some(list) => list,
        None => {
            panic!();
        }
    };

    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 and ACKed by master
/// `container_tx`: mpsc Sender used pass recieved [ElevatorContainer]'s 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<WorldView>,
    remove_container_tx: mpsc::Sender<u8>,
    container_tx: mpsc::Sender<ElevatorContainer>,
    connection_to_master_failed_tx: mpsc::Sender<bool>,
    sent_tcp_container_tx: mpsc::Sender<ElevatorContainer>,
    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;

        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;
    }
}



/* __________ END PUBLIC FUNCTIONS __________ */








/* __________ START PRIVATE FUNCTIONS __________ */
```

```
/// Function to read TcpStream to a slave
///
/// # Parameters
/// `stream`: The stream connected to the slave
/// `remove_container_tx`: mpsc Sender used to notify worldview updater if a slave should be removed
/// `container_tx`: mpsc Sender used pass recieved [ElevatorContainer] to the worldview_updater
///
/// # Behavior
/// The function continously reads from the stream, and sends recieved [ElevatorContainer]'s on `container_tx`.
async fn start_reading_from_slave(
    mut stream: TcpStream,
    remove_container_tx: mpsc::Sender<u8>,
    container_tx: mpsc::Sender<ElevatorContainer>
) {
    loop {
        /* Tries to read from stream */
        let result = read_from_stream(remove_container_tx.clone(), &mut stream).await;
        match result {
            Some(msg) => {
                let _ = container_tx.send(msg).await;
            }
            None => {
                break;
            }
        }

    }
}


/// Function that handles TCP while you are master on the system
///
/// # Parameters
/// `wv`: A mutable refrence to the current 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 [ElevatorContainer] 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 runs [start_reading_from_slave] on the stream
async fn tcp_while_master(
    wv: &mut WorldView,
    wv_watch_rx: watch::Receiver<WorldView>,
    socket_rx: &mut mpsc::Receiver<(TcpStream, SocketAddr)>,
    remove_container_tx: mpsc::Sender<u8>,
    container_tx: mpsc::Sender<ElevatorContainer>
) {
    /* While you are master */
    while world_view::is_master(&wv) {
        /* 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 {
                    /* Start handling the slave. Also has watchdog function to detect timeouts on messages */
                    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 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 WorldView,
    wv_watch_rx: watch::Receiver<WorldView>,
    connection_to_master_failed_tx: mpsc::Sender<bool>,
    sent_tcp_container_tx: mpsc::Sender<ElevatorContainer>
) {
    /* 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()).await {
        println!("Master accepted the TCP-connection");
        master_accepted_tcp = true;
        stream = Some(s);
    } else {
        println!("Master did not accept the TCP-connection");
        sleep(Duration::from_secs(100)).await;
        let _ = connection_to_master_failed_tx.send(true).await;
    }

    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) && 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).await;
                if new_master {
                    print::slave(format!("New master on the network"));
                    master_accepted_tcp = false;
                    let _ = sleep(config::SLAVE_TIMEOUT);
                }
                prev_master = wv.master_id;
                world_view::update_wv(wv_watch_rx.clone(), wv).await;
                if prev_master != wv.master_id {
                    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
///
/// # 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.
async fn connect_to_master(
    wv_watch_rx: watch::Receiver<WorldView>
) -> 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.master_id, config::PN_PORT);
        println!("Master id: {}", wv.master_id);
        print::info(format!("Trying to connect to : {} in connect_to_master()", master_ip));

        let socket = match create_tcp_socket() {
            Ok(sock) => {
                sock
            },
            Err(e) => {
                print::err(format!("Error while creating tcp-socket for connecting to master: {}", e));
                return None;
            }
        };

        return connect_socket(socket, &master_ip);
    } else {
        None
    }
}



/// 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 [ElevatorContainer] 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 sends an ACK on the stream, and 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<ElevatorContainer> {
    let id = ip_help_functions::ip2id(stream.peer_addr().expect("Slave has no IP?").ip());
    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 _ =  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 _ =  remove_container_tx.send(id).await;
                            return None;
                        }
                        Ok(_) => {
                            //TODO: ikke let _ =
                            let _ =  stream.write_all("ACK".as_bytes()).await;
                            let _ = stream.flush().await;

                            return world_view::deserialize(&buffer)

                        },
                        Err(e) => {
                            print::err(format!("Error while reading from stream: {}", e));
                            let _ =  remove_container_tx.send(id).await;
                            return None;
                        }
                    }
                }
                Err(e) => {
                    print::err(format!("Error while reading from stream: {}", e));
                    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;
        }
    }
}


/// 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
///
/// # 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 reads one byte from stream (used as ACKing from master)
/// Once the ACK is recieved, it sends the sent data over `sent_tcp_container_tx`.
/// If writing to or reading from 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<ElevatorContainer>,
    stream: &mut TcpStream,
    wv: &WorldView
) {
    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 = world_view::serialize(&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 mut buf: [u8; 3] = [0, 0, 0];
        match stream.read_exact(&mut buf).await {
            Ok(_) => {
                // println!("Master acka: {:?}", buf.to_ascii_uppercase());
                let _ = sent_tcp_container_tx.send(self_elev_container.clone()).await;
            },
            Err(e) => {
                print::err(format!("Master did not ACK the message: {}", e));
                let _ = connection_to_master_failed_tx.send(true).await;
            }
        }
    }
}
```

```
/// Creates a `TcpListener` from a given socket and binds it to the system's network address.
///
/// This function constructs a `SocketAddr` using the system's network prefix, the device's self ID,
/// and the configured port. It then binds the socket to this address and starts listening for incoming
/// TCP connections.
///
/// # Arguments
/// * `socket` - The `Socket` instance to use for listening.
///
/// # Returns
/// * `Some(TcpListener)` - If binding and listening are successful.
/// * `None` - If an error occurs.
///
/// # Errors
/// * If the system's network address cannot be created, an error is printed and `None` is returned.
/// * If binding the socket to the address fails, an error is printed and `None` is returned.
/// * If listening on the socket fails, an error is printed and `None` is returned.
/// * If converting the socket into a `TcpListener` fails, an error is printed and `None` is returned.
fn listener_from_socket(
    socket: Socket
) -> Option<TcpListener> {
    // Attempts to parse the socket address to self_ip
        let self_ip: SocketAddr = match format!("{}.{}:{}", config::NETWORK_PREFIX, network::read_self_id(),
config::PN_PORT).parse() {
        Ok(addr) => addr,
        Err(e) => {
            print::err(format!("Failed to setup self listener socketaddr: {}", e));
            return None;
        }
    };

    // Attemps to bind the socket to self_ip
    match socket.bind(&self_ip.into()) {
        Ok(_) => {},
        Err(e) => {
            print::err(format!("Failed to bind socket: {}", e));
            return None
        }
    }

    // Attemps to start listening on the socket
    match socket.listen(128) {
        Ok(_) => {},
        Err(e) => {
            print::err(format!("Failed to start listening: {}", e));
            return None;
        }
    }

    // Set non blocking, so it can be used by Tokio
    socket.set_nonblocking(true).expect("Coulndt set socket non-blocking");
    // Convert the socket to an asynchrounus TcpListener
    match TcpListener::from_std(socket.into()) {
        Ok(listener) => {
            print::ok(format!("System listening on {}:{}", self_ip, config::PN_PORT));
            return Some(listener);
        },
        Err(e) => {
            print::err(format!("Failed to parse socket to tcplistener: {}", e));
            return None;
        }
    };
}


/// Attempts to connect a given socket to a specified target address.
///
/// This function takes a `Socket` and a `target` string, parses the string into a `SockAddr`,
/// and tries to establish a TCP connection. If successful, it converts the socket into a `TcpStream`
/// and returns it wrapped in an `Option`.
///
/// # Arguments
/// * `socket` - The `Socket` instance to connect.
/// * `target` - A `String` containing the target IP address and port in the format "IP:PORT".
///
/// # Returns
/// * `Some(TcpStream)` - If the connection is successful.
/// * `None` - If an error occures.
///
/// # Errors
/// * If `target` cannot be parsed into a valid `SocketAddr`, an error is printed and `None` is returned.
/// * If the connection attempt fails, an error is printed and `None` is returned.
/// * If converting the socket into a `TcpStream` fails, an error message is printed and `None` is returned.
fn connect_socket(
    socket: Socket,
    target: &String
) -> Option<TcpStream> {
    // Attempts to parse the address
    let master_sock_addr: SockAddr = match target.parse::<SocketAddr>() {
        Ok(addr) => SockAddr::from(addr),
        Err(e) => {
            print::err(format!("Failed to parse string: {} into address: {}", target, e));
            return None;
        }
    };

    // Attempts to connect the socket to the destination address
    match socket.connect(&master_sock_addr) {
        Ok(()) => {
            print::ok(format!("Connected to Master: {} i TCP_listener()", target));
        },
        Err(e) => {
            print::err(format!("Failed to connect to master: {}", e));
            return None;
        },
    };

    // Set non blocking, so it can be used by Tokio
    socket.set_nonblocking(true).expect("Coulndt set socket non-blocking");

    // Convert the socket into a standard TcpStream
    let std_stream: std::net::TcpStream = socket.into();

    // Convert the standard TcpStream to an asynchrounus tokio TcpStream
    match TcpStream::from_std(std_stream) {
        Ok(stream) => {
            return Some(stream);
        },
        Err(e) => {
            eprintln!("Failed to convert socket to TcpStream: {}", e);
            return None;
        }
    };
}

/// Creates and configures a TCP socket with optimized settings for performance and reliability.
///
/// The function sets buffer sizes, keepalive settings, timeouts, and platform-specific options.
///
/// # Returns
/// * `Ok(Socket)` - A configured TCP socket.
/// * `Err(Error)` - If socket creation or configuration fails.
///
/// # Platform-Specific Behavior
/// * Some options, such as `set_thin_linear_timeouts`, `set_tcp_user_timeout`, `set_quickack`, and `set_cork`,
///   are only available on Linux and will not be set on Windows.
fn create_tcp_socket() -> Result<Socket, Error> {
    // Create a new TCP socket
    let socketres = Socket::new(Domain::IPV4, Type::STREAM, Some(Protocol::TCP));
    let socket: Socket = match socketres {
        Ok(sock) => sock,
        Err(e) => {return Err(e);},
    };

    // Set send and receive buffer sizes to 16MB for high-throughput connections.
    socket.set_send_buffer_size(16_777_216)?;
    socket.set_recv_buffer_size(16_777_216)?;

    // Configure TCP keepalive to detect broken connections.


    #[cfg(target_os = "linux")]
    {
        let keepalive = TcpKeepalive::new()
            .with_time(Duration::from_secs(10))         // Start sending keepalive probes after 10 seconds of inactivity.
            .with_interval(Duration::from_secs(1))                  // Send keepalive probes every 1 second.
            // On Linux, specify the number of keepalive probes before the connection is considered dead.
            // This setting is not available on Windows.
            .with_retries(10);
        socket.set_tcp_keepalive(&keepalive.to_owned().clone())?;
    }

    #[cfg(target_os = "windows")]
    {
        let keepalive = TcpKeepalive::new()
            .with_time(Duration::from_secs(10))         // Start sending keepalive probes after 10 seconds of inactivity.
            .with_interval(Duration::from_secs(1));                  // Send keepalive probes every 1 second.
        socket.set_tcp_keepalive(&keepalive.to_owned().clone())?;
    }

    // Set read and write timeouts to 10 seconds.
    socket.set_read_timeout(Some(Duration::from_secs(30)))?;
    socket.set_write_timeout(Some(Duration::from_secs(20)))?;

    #[cfg(target_os = "linux")]
    {
        // Enable thin linear timeouts (Linux only), drastically improving retransmission timing under congestion.
        socket.set_thin_linear_timeouts(true)?;

        // Set TCP user timeout (Linux only) to close the connection if no acknowledgments are received within 10s.
        socket.set_tcp_user_timeout(Some(Duration::from_secs(20)))?;

        // Enable TCP Quick ACK (Linux only), reducing latency by immediately acknowledging received packets.
        socket.set_quickack(true)?;
    }

    // Disable Nagles algorithm to minimize latency for interactive or real-time applications.
    // This ensures small packets are sent immediately instead of being buffered.
    socket.set_nodelay(true)?;

    Ok(socket)
}


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

Innhald frå Rust-filer

Fil: elevator_pro_rebrand\src\network\udp_net.rs

```
use crate::config;
use crate::ip_help_functions;
use crate::network;
use crate::print;
use crate::world_view;
use crate::world_view::ElevatorContainer;
use crate::world_view::WorldView;

use tokio::time::sleep;
use tokio::net::UdpSocket;
use socket2::{Domain, Socket, Type, Protocol};
use tokio::sync::{watch, mpsc, Mutex};
use std::net;
use std::{
    collections::HashMap,
    net::{SocketAddr, Ipv4Addr},
    sync::Arc,
    time::{Duration, Instant},
};
use std::sync::atomic::{AtomicBool, Ordering};


const INACTIVITY_TIMEOUT: Duration = Duration::from_secs(5); // Tidsgrense for inaktivitet
const CLEANUP_INTERVAL: Duration = Duration::from_secs(1); // Hvor ofte inaktive sendere fjernes

pub static IS_MASTER: AtomicBool = AtomicBool::new(false);


pub async fn start_udp_network(
    wv_watch_rx: watch::Receiver<WorldView>,
    container_tx: mpsc::Sender<ElevatorContainer>,
    packetloss_rx: watch::Receiver<network::ConnectionStatus>,
    connection_to_master_failed_tx: mpsc::Sender<bool>,
    remove_container_tx: mpsc::Sender<u8>,
    sent_tcp_container_tx: mpsc::Sender<ElevatorContainer>,
) {
    while !network::read_network_status() {}
    let socket = match Socket::new(Domain::IPV4, Type::DGRAM, Some(Protocol::UDP)) {
        Ok(sock) => sock,
        Err(e) => {panic!("Klarte ikke lage udp socket");}
    };
    while socket.set_reuse_address(true).is_err() {}
    while socket.set_send_buffer_size(16_000_000).is_err() {}
    while socket.set_recv_buffer_size(16_000_000).is_err() {}

    let addr: SocketAddr = format!("{}.{}:{}", config::NETWORK_PREFIX, network::read_self_id(), 50000).parse().unwrap();

    while socket.bind(&addr.into()).is_err() {}
    while socket.set_nonblocking(true).is_err() {}

    let socket = match UdpSocket::from_std(socket.into()) {
        Ok(sock) => sock,
        Err(e) => {panic!("Klarte ikke lage tokio udp socket");}
    };


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

        IS_MASTER.store(false, Ordering::SeqCst);
        send_udp_slave(
            &socket,
            &mut wv,
            wv_watch_rx.clone(),
            packetloss_rx.clone(),
            connection_to_master_failed_tx.clone(),
            sent_tcp_container_tx.clone(),
        ).await;
    }
}



/// Holder informasjon om en aktiv sender
#[derive(Debug, Clone)]
struct ReceiverState {
    last_seq: u16,
    last_seen: Instant,
}

/// Starter en UDP-mottaker som håndterer flere samtidige sendere og sender redundante ACKs
async fn receive_udp_master(
    socket: &UdpSocket,
    wv: &mut WorldView,
    wv_watch_rx: watch::Receiver<WorldView>,
    container_tx: mpsc::Sender<ElevatorContainer>,
    packetloss_rx: watch::Receiver<network::ConnectionStatus>,
    remove_container_tx: mpsc::Sender<u8>,
) {
    world_view::update_wv(wv_watch_rx.clone(), wv).await;
    println!("Server listening on port {}", 50000);
```

```
let state = Arc::new(Mutex::new(HashMap::<SocketAddr, ReceiverState>::new()));

// Cleanup-task: Fjerner inaktive klienter
let state_cleanup = state.clone();
{
    let wv_watch_rx = wv_watch_rx.clone();
    let mut wv = wv.clone();
    tokio::spawn(async move {
        while wv.master_id == network::read_self_id() {
            sleep(CLEANUP_INTERVAL).await;
            {
                let mut state = state_cleanup.lock().await;
                let now = Instant::now();

                //Remove inactive slaves, save SocketAddr to the removed ones
                for (adr, stat) in state.iter() {
                    println!("Addr: {:?}, Time: {:?}", adr.ip(), Instant::now() - stat.last_seen);
                }
                let mut removed = Vec::new();
                state.retain(|k, s| {
                    let keep = now.duration_since(s.last_seen) < INACTIVITY_TIMEOUT;
                    if !keep {
                        removed.push(*k);
                    }
                    keep
                });


                // Send the ID of the removed slaves to worldview updater
                for addr in removed {
                    let _ = remove_container_tx.send(ip_help_functions::ip2id(addr.ip())).await;
                }
            }
            world_view::update_wv(wv_watch_rx.clone(), &mut wv).await;
        }
    });
}

let mut buf = [0; 65535];
while wv.master_id == network::read_self_id() {
    // println!("min id: {}, master ID: {}", network::read_self_id(), wv.master_id);
    // Mottar data
    let (len, slave_addr) = match socket.try_recv_from(&mut buf) {
        Ok(res) => res,
        Err(ref e) if e.kind() == std::io::ErrorKind::WouldBlock => {
            // Egen case for når bufferet er tomt
            sleep(config::POLL_PERIOD).await;
            world_view::update_wv(wv_watch_rx.clone(), wv).await;
            continue;
        }
        Err(e) => {
```

```
        eprintln!("Error receiving UDP packet: {}", e);
        world_view::update_wv(wv_watch_rx.clone(), wv).await;
        continue;
    }
};

let mut new_state = ReceiverState {
    last_seq: 0,
    last_seen: Instant::now(),
};

let mut state_locked = state.lock().await;
let entry = state_locked.entry(slave_addr).or_insert(new_state.clone());
let last_seen = entry.last_seen;
let last_seq = entry.last_seq.clone();

let msg = parse_message(&buf[..len], last_seq);

match msg {
    (Some(container), code) => {
        // println!("Received valid packet from {}: seq {}", slave_addr, last_seq);
        //Meldinga er en forventet melding -> oppdater hashmappets state
        // println!("Ack? {:?}", code);
        match code {
            RecieveCode::Accept | RecieveCode::Rejoin=> {
                let _ = container_tx.send(container.clone()).await;
                new_state.last_seq = last_seq.wrapping_add(1);
                if code == RecieveCode::Rejoin {
                    new_state.last_seq = 0;
                }
                new_state.last_seen = Instant::now();
                state_locked.insert(slave_addr, new_state);

            },
            RecieveCode::AckOnly => {},
            RecieveCode::Ignore => {},
        }

        if code != RecieveCode::Ignore {
            let packetloss = packetloss_rx.borrow().clone();
                let redundancy = get_redundancy(packetloss.packet_loss, last_seen).await;
                print::warn(format!(
                    "Sending {} ACKs to {} (loss: {}%, time since last: {:.2}s)",
                    redundancy,
                    slave_addr,
                    packetloss.packet_loss,
                    Instant::now().duration_since(last_seen).as_secs_f64()
                ));
                send_acks(
                    &socket,
                    last_seq,
                    &slave_addr,
                        redundancy
                    ).await;
                }
            },
            (None, _) => {
                // println!("Ignoring out-of-order packet from {}", slave_addr);
                // Seq nummer doesnt match, or data has been corrupted.
                // Treat it as if nothing was read.
                //TODO: Should update last instant? maybe not in case seq number gets unsynced?
            }
        }
        world_view::update_wv(wv_watch_rx.clone(), wv).await;
    }
}


async fn send_acks(
    socket: &UdpSocket,
    seq_num: u16,
    addr: &SocketAddr,
    redundancy: usize
) {
    for _ in 0..redundancy {
        let data = seq_num.to_le_bytes();
        let _ = socket.send_to(&data, addr).await;
    }
}



async fn send_udp_slave(
    socket: &UdpSocket,
    wv: &mut WorldView,
    wv_watch_rx: watch::Receiver<WorldView>,
    packetloss_rx: watch::Receiver<network::ConnectionStatus>,
    connection_to_master_failed_tx: mpsc::Sender<bool>,
    sent_tcp_container_tx: mpsc::Sender<ElevatorContainer>,
) {
    world_view::update_wv(wv_watch_rx.clone(), wv).await;
    let mut seq = 0;
    while wv.master_id != network::read_self_id() {
        world_view::update_wv(wv_watch_rx.clone(), wv).await;
        while send_udp(socket, wv, packetloss_rx.clone(), 50, seq, 20, sent_tcp_container_tx.clone()).await.is_err() {
            let _ = connection_to_master_failed_tx.send(true).await;
            sleep(config::SLAVE_TIMEOUT).await;
            world_view::update_wv(wv_watch_rx.clone(), wv).await;
            return;
        }
        seq = seq.wrapping_add(1);
        sleep(config::SLAVE_TIMEOUT).await;
    }
}
```

```
async fn send_udp(
    socket: &UdpSocket,
    wv: &WorldView,
    packetloss_rx: watch::Receiver<network::ConnectionStatus>,
    timeout_ms: u64,
    seq_num: u16,
    retries: u16,
    sent_tcp_container_tx: mpsc::Sender<ElevatorContainer>,
)  -> std::io::Result<()> {

    // Må sikre at man er online
    // TODO: Send inn ferdig binda socket, den kan heller lages i slave_loopen!

    let server_addr: SocketAddr = format!("{}.{}:{}", config::NETWORK_PREFIX, wv.master_id, 50000).parse().unwrap();
    let mut buf = [0; 65535];

    let mut last_seen_from_master = Instant::now();

    let mut fails = 0;
    let mut backoff_timeout_ms = timeout_ms;

    let mut should_send: bool = true;
    let sent_cont = match world_view::extract_self_elevator_container(wv) {
        Some(cont) => cont.clone(),
        None => {
            return Err(std::io::Error::new(std::io::ErrorKind::InvalidData, "Self container not found in worldview"))
        }
    };
    let mut timeout = sleep(Duration::from_millis(backoff_timeout_ms));
    loop {
        if should_send {
            let packetloss = packetloss_rx.borrow().clone();
            let redundancy = get_redundancy(packetloss.packet_loss, last_seen_from_master).await;
            // println!(
            //     "Sending packet {} with redundancy {} (loss: {}%, time since last ACK: {:.2}s)",
            //     seq_num,
            //     redundancy,
            //     packetloss.packet_loss,
            //     Instant::now().duration_since(last_seen_from_master).as_secs_f64()
            // );
            println!("Sending with redundancy: {}", redundancy);
                        // println!("Sending packet nr. {} with {} copies (estimated loss: {}%)", seq_num, redundancy,
packetloss.packet_loss);
            send_packet(
                &socket,
                seq_num,
                &server_addr,
                redundancy,
                &wv
            ).await?;
            backoff_timeout_ms += 5;
            should_send = false;
        }

        timeout = sleep(Duration::from_millis(backoff_timeout_ms));

        // Add 10 ms timeout for each retransmission.
        // In a real network: should probably be exponential.
        // In Sanntidslabben: Packetloss is software, slow ACKs is packetloss, not congestion or long travel links.
        // The only reason this is added here is because the new script (which doesnt work) has an option for latency.
        // backoff_timeout_ms += 5;
        tokio::select! {
            _ = timeout => {
                fails += 1;
                println!(
                    "Timeout (seq: {}, dest: {}). Retransmitting attempt {}/{}...",
                    seq_num, server_addr, fails, retries
                );
                if fails > retries {
                        return Err(std::io::Error::new(std::io::ErrorKind::TimedOut, format!("No Ack from master in {} retries!",
retries)));
                }
                should_send = true;
            },
            result = socket.recv_from(&mut buf) => {
                if let Ok((len, addr)) = result {
                    let seq_opt: Option<[u8; 2]> = buf[..len].try_into().ok();
                    if let Some(seq) = seq_opt {
                        if seq_num == u16::from_le_bytes(seq) {
                            last_seen_from_master = Instant::now();
                            let _ = sent_tcp_container_tx.send(sent_cont).await;
                            println!("Master acked the cont");
                            return Ok(());
                        }
                    }
                    // Hvis pakken ikke var riktig ACK, fortsett til neste forsøk.
                }
            },
        }

    }
}


// fn get_redundancy(packetloss: u8) -> usize {
//     match packetloss {
//         // p if p < 25 => 4,
//         // p if p < 50 => 8,
//         // p if p < 75 => 16,
//         // p if p < 90 => 20,
//         _ => 100,
//     }
// }



async fn send_packet(
    socket: &UdpSocket,
    seq_num: u16,
    addr: &SocketAddr,
    redundancy: usize,
    wv: &WorldView
) -> std::io::Result<()> {
    let data_opt = build_message(wv, &seq_num);
    if let Some(data) =  data_opt {
        for _ in 0..redundancy {
            let _ = socket.send_to(&data, addr).await;
        }
        return Ok(())
    } else {
        return Err(std::io::Error::new(std::io::ErrorKind::Other, "Failed to build UDP message to master"));
    }
}


fn build_message(
    wv: &WorldView,
    seq_num: &u16,
) -> Option<Vec<u8>> {
    let mut buf = Vec::new();

    let seq = seq_num.to_le_bytes();
    buf.extend_from_slice(&seq);


    let cont = world_view::extract_self_elevator_container(&wv)?;

    let ec_bytes = world_view::serialize(&cont);
    buf.extend_from_slice(&ec_bytes);

    Some(buf)
}

fn parse_message(
    buf: &[u8],
    expected_seq: u16,
) -> (Option<ElevatorContainer>, RecieveCode) {
    if buf.len() < 2 {
        return (None, RecieveCode::Ignore);
    }


    let seq: [u8; 2] = match buf[0..2].try_into().ok() {
        Some(number) => number,
        None => return (None, RecieveCode::Ignore),
    };
    let key = u16::from_le_bytes(seq);

    if key == expected_seq {
        return (world_view::deserialize(&buf[2..]), RecieveCode::Accept);
    } else if key == 0 && expected_seq != 0 {
        return (world_view::deserialize(&buf[2..]), RecieveCode::Rejoin);
    } else if key == expected_seq.wrapping_rem(1) {
        return (world_view::deserialize(&buf[2..]), RecieveCode::AckOnly);
    } else {
        return (None, RecieveCode::Ignore);
    }
}


#[derive(Debug, Clone, PartialEq)]
enum RecieveCode {
    Accept,
    AckOnly,
    Ignore,
    Rejoin
}


struct PID {
    kp: f64,
    ki: f64,
    kd: f64,
    prev_error: f64,
    integral: f64,
    last_time: Option<Instant>,
}

impl PID {
    fn new(kp: f64, ki: f64, kd: f64) -> Self {
        Self {
            kp,
            ki,
            kd,
            prev_error: 0.0,
            integral: 0.0,
            last_time: None,
        }
    }

    fn update(&mut self, setpoint: f64, measurement: f64, now: Instant) -> f64 {
        let error = -(setpoint - measurement);
        let dt = self.last_time.map_or(0.1, |last| {
            let secs = now.duration_since(last).as_secs_f64();
            if secs < 0.001 { 0.001 } else { secs }
        });
        self.integral += clamp(error * dt, -20.0, 20.0);
        let derivative = (error - self.prev_error) / dt;
        self.prev_error = error;
        self.last_time = Some(now);

        self.kp * error + self.ki * self.integral + self.kd * derivative
    }
}

use once_cell::sync::Lazy;

static REDUNDANCY_PID: Lazy<Mutex<PID>> = Lazy::new(|| {
    Mutex::new(PID::new(60.0, 14.05, 1.01)) // Tuning-verdiar: test gjerne!
});

fn clamp(val: f64, min: f64, max: f64) -> f64 {
    val.max(min).min(max)
}

pub async fn get_redundancy(packetloss: u8, last_seen: Instant) -> usize {
    let now = Instant::now();
    let time_since_last = now.duration_since(last_seen).as_secs_f64(); // i sekund

    let setpoint = 0.1; // 10 ms ønsket tid mellom mottak
    let measurement = time_since_last;

    let output = {
        let mut pid = REDUNDANCY_PID.lock().await;
        pid.update(setpoint, measurement, now)
    };

    let base = 1.0;
    let redundans = clamp((base + output)*(packetloss as f64+1.0)/100.0, 1.0, 300.0);

    // println!(
    //     "[PID] Last seen: {:.3}s | Error: {:.3} | Redundancy: {:.1}",
    //     time_since_last,
    //     setpoint - measurement,
    //     redundans
    // );

    redundans.round() as usize
}
```

Innhald frå Rust-filer

Fil: elevator_pro_rebrand\src\network\udp_network.rs

```rust
//! ## UDP Module
//!
//! This module handles UDP communication in the network. It is responsible for broadcasting the worldview when acting as the master
//! and listening for worldview broadcasts when acting as a slave. The module ensures that received broadcasts originate from the expected network.
//!
//! ## Overview
//! - **Master Node**: Broadcasts the current worldview on a UDP channel.
//! - **Slave Node**: Listens for worldview broadcasts from the network master and updates its state accordingly.
//! - **UDP Watchdog**: Detects timeouts when no valid broadcasts are received.
//!
//! ## Key Features
//! - Uses a reusable UDP socket for broadcasting and listening.
//! - Ensures messages are from the correct network by checking a predefined key string.
//! - Implements a watchdog mechanism to detect loss of connection to the master.
//!
//! ## Functions
//!
//! - [`start_udp_broadcaster`]: Sends worldview data over UDP if this node is the master.
//! - [`start_udp_listener`]: Listens for worldview broadcasts from the master and updates state.
//! - [`udp_watchdog`]: Detects connection loss to the master by monitoring broadcast activity.
//! - Private helper functions: [`get_udp_timeout`], [`build_message`], [`parse_message`].
//!
//! ## Usage
//! These functions should be called asynchronously in a Tokio runtime.

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

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 tokio::sync::watch;

use super::local_network;

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


/* __________ START PUBLIC FUNCTIONS __________ */
```

```rust
// ### 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<WorldView>
) -> tokio::io::Result<()> {
    while !network::read_network_status() {

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

    // Set up 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("Invalid address"); // UDP-broadcast address
    let socket_addr: SocketAddr = addr2.parse().expect("Invalid address");
    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;
        // If you currently are master on the network
        if network::read_self_id() == wv.master_id {
            sleep(config::UDP_PERIOD);
            let message_bytes = build_message(&wv);

            // If you are connected to internet
            if network::read_network_status() {
                // If you also were connected to internet last time you ran this
                if !prev_network_status {
                    sleep(Duration::from_millis(500));
                    prev_network_status = true;
                }
                // Send your worldview on UDP broadcast
                match udp_socket.send_to(&message_bytes, &broadcast_addr).await {
                    Ok(_) => {
                        // print::ok(format!("Sent udp broadcast!"));
                    },
                    Err(_) => {
                        // print::err(format!("Error while sending UDP: {}", e));
                    }
                }

            }else {
                prev_network_status = false;
            }
        }
    }
}

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

    }
    //Set up 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("Invalid address");
    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: Option<WorldView>;
    let mut my_wv = world_view::get_wv(wv_watch_rx.clone());

    loop {
        // Read message on UDP-broadcast address
        match socket.recv_from(&mut buf).await {
            Ok((len, _)) => {
                read_wv = parse_message(&buf[..len]);
            }
            Err(e) => {
                return Err(e);
            }
        }

        match read_wv {
            Some(mut read_wv) => {
                world_view::update_wv(wv_watch_rx.clone(), &mut my_wv).await;

                if read_wv.master_id != my_wv.master_id {
                    // Ignore
                } else {
                    // The message came from the current master -> reset the watchdog
                    get_udp_timeout().store(false, Ordering::SeqCst);
                }

                // Pass the recieved WorldView if the message came from the master or a node with a lower ID than current
master,
                // and this node is not the master
                if my_wv.master_id >= read_wv.master_id
                    && self_id != read_wv.master_id
                {
                    my_wv = read_wv;
                    let _ = udp_wv_tx.send(my_wv.clone()).await;
                }
            },
            None => continue,
        }
    }
}


/// 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 || !network::read_network_status(){
            get_udp_timeout().store(true, Ordering::SeqCst);
            tokio::time::sleep(Duration::from_millis(5000)).await;
        }
        else {
            get_udp_timeout().store(false, Ordering::SeqCst);
            print::warn("UDP-watchdog: Timeout".to_string());
            let _ = connection_to_master_failed_tx.send(true).await;
        }
    }
}

/* __________ END PUBLIC FUNCTIONS __________ */




/* __________ START PRIVATE FUNCTIONS __________ */

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

/// Builds the UDP-broadcast message from the worldview
///
/// # Parameters
/// `wv`: Reference to the current [WorldView]
///
/// # Returns
/// -`Vec<u8>`: Containing serialized data of the message, ready to be sent
///
/// # Behavior
/// The function serializes a key, used for other nodes on the network to recognize this broadcast from others,
/// and appends the serialized data of the worldview.
fn build_message(
    wv: &WorldView
) -> Vec<u8> {
    let mut buf = Vec::new();

    // Add the serialized key
    let key_bytes = world_view::serialize(&config::KEY_STR);
    buf.extend_from_slice(&key_bytes);

    // Add the serialized worldview
    let wv_bytes = world_view::serialize(&wv);
    buf.extend_from_slice(&wv_bytes);

    buf
}

/// Reconstructs a [WorldView] from recieved UDP-message
///
/// # Parameters
/// `buf`: Referance to a buffer containing the raw data read from UDP
///
/// # Returns
/// -`Option<WorldView>`: A WorldView reconstructed from the data, if no errors occures
/// -`None`: If an error occures while deserializing, or if the broadcast does not contain our key
///
/// # Behavior
/// The function first looks for the [config::KEY_STR] in the beginning og the message, returning `None` if it is not found.
/// If it is found, the function tries to deserialize a [WorldView] from the rest of the message, returning it wrapped in an `Option` if it succeeded, returning `None` if it failed.
pub fn parse_message(
    buf: &[u8]
) -> Option<WorldView> {
    // 1. Prøv å deserialisere nøkkelen
    let key_len = bincode::serialized_size(config::KEY_STR).unwrap() as usize;

    if buf.len() <= key_len {
        return None;
    }

    let (key_part, wv_part) = buf.split_at(key_len);

    let key: String = bincode::deserialize(key_part).ok()?;
    if key != config::KEY_STR {
        return None; // feil nøkkel
    }

    // 2. Deserialize resten til WorldView
    world_view::deserialize(wv_part)
}


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

Innhald frå Rust-filer

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

```rust
//! # World View Channel Handler
//!
//! This module handles messages on internal MPSC channels related to updates of the shared `WorldView`.
//!
//! It acts as the local node's **central synchronization point**, receiving structured messages from:
//! - The UDP listener (initial worldview from network)
//! - The TCP slave/master communication (container updates and disconnections)
//! - The local elevator state machine
//! - The task manager (e.g. delegated tasks)
//!
//! ## Responsibilities
//! - Maintain and update the local node's view of the system state (`WorldView`).
//! - Apply state changes to the elevators own container (e.g., door, obstruction).
//! - Integrate updates from other elevators via serialized containers.
//! - Ensure consistency between local memory and the distributed worldview.
//!
//! ## Structure
//! The module consists of:
//! - `update_wv_watch(...)`: the main loop that listens on MPSC channels and updates the shared `WorldView`.
//! - `Mpscs`: a struct bundling all the MPSC senders and receivers used to exchange data between modules.
//!
//! ## Design Considerations
//! - This module separates concerns between **event reception** (via channels) and **data transformation** (in `world_view_update`).
//! - The logic is intentionally asynchronous and non-blocking, reflecting the concurrent nature of real-time elevator systems.
//!
//! ## Access Pattern
//! This module is called once per process as part of system initialization and runs in its own async task.
//! It ensures that the shared state remains up to date and consistent across elevator roles (master/slave).

mod update_wv;

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

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};

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.

/// Continuously updates the local `WorldView` based on system events and communication channels.
///
/// This function is the central synchronization loop for each elevator node. It listens to a range
/// of MPSC channels and applies changes to the shared `WorldView` structure accordingly. The updated
/// worldview is then sent through a `watch` channel to propagate state to other modules or tasks.
///
/// # Parameters
/// - `mpsc_rxs`: A struct containing all MPSC receiver channels used to receive events related to worldview changes.
/// - `worldview_watch_tx`: A watch channel sender used to broadcast updated copies of the worldview to subscribers.
/// - `worldview`: A mutable reference to the current local worldview instance.
///
/// # Behavior
/// The function operates as an infinite loop, continuously polling the following channels:
///
/// ### Slave-related channels:
/// - `sent_tcp_container`: Removes tasks or hall requests that were successfully transmitted to master.
/// - `udp_wv`: Merges received worldview via UDP (usually at startup or reconnection).
/// - `connection_to_master_failed`: Triggers "network abort", clearing out all elevators except the local one.
///
/// ### Master-related channels:
/// - `container`: Updates worldview with elevator data received from a slave.
/// - `remove_container`: Removes a disconnected elevator from the worldview.
/// - `delegated_tasks`: Updates each elevators task list with assignments from the task allocator.
///
/// ### Shared (master/slave):
/// - `elevator_states`: Updates the local elevator container with new status values (e.g., door open, obstruction).
/// - `new_wv_after_offline`: Merges two worldviews when reconnecting to the network after being offline.
///
/// After applying updates, the function broadcasts the new worldview using `worldview_watch_tx`.
/// If the current elevator is the master, its updated container is also looped back into the update process
/// to simulate its own TCP contribution.
///
/// # Critical Role
/// This function is essential for the functioning of the distributed system. Without it,
/// the local node will not respond to state updates, new tasks, disconnections, or network changes.
///
/// It must be run as an asynchronous task during system startup and should never exit during runtime.
#[allow(non_snake_case)]
pub async fn update_wv_watch(mut mpsc_rxs: MpscRxs, worldview_watch_tx: watch::Sender<WorldView>, mut
worldview: &mut WorldView) {
    let _ = worldview_watch_tx.send(worldview.clone());
    let mut wv_edited_I = false;
    let mut master_container_updated_I = false;

    let (master_container_tx, mut master_container_rx) = mpsc::channel::<ElevatorContainer>(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, msg);
            },
            Err(_) => {},
        }
        /*_____Update worldview based on worldviews recieved on UDP_____ */
        match mpsc_rxs.udp_wv.try_recv() {
            Ok(mut master_wv) => {
                wv_edited_I = join_wv_from_udp(&mut worldview, &mut 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);
            },
            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, &container).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, &container).await;
            },
            Err(_) => {},
        }
        /*_____Update worldview when a slave should be removed_____ */
        match mpsc_rxs.remove_container.try_recv() {
            Ok(id) => {
                println!("Skal fjerne ID: {}", id);
                wv_edited_I = remove_container(&mut worldview, 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, 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, container);
                master_container_updated_I = world_view::is_master(&worldview);
            },
            Err(_) => {},
        }
        /*_____Update worldview after you reconeccted to internet  */
        match mpsc_rxs.new_wv_after_offline.try_recv() {
            Ok(mut read_wv) => {
                merge_wv_after_offline(&mut worldview, &mut read_wv);
                let _ = worldview_watch_tx.send(worldview.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) {
                let _ = master_container_tx.send(container.clone()).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.clone());
            wv_edited_I = false;
        }
    }
}


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

```rust
/// 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<WorldView>,

    /// 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<ElevatorContainer>,

    /// 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<ElevatorContainer>,

    /// Sends delegated tasks from the manager
    pub delegated_tasks: mpsc::Sender<HashMap<u8, Vec<[bool; 2]>>>,

    /// Send ElevatorContainer from the local elevator handler
    pub elevator_states: mpsc::Sender<ElevatorContainer>,

    /// Sends the new worldview after reconnecting to the network
    pub new_wv_after_offline: mpsc::Sender<WorldView>,

    /// Additional buffer channels that can be used if necessary
    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 {
    /// Recieves a UDP worldview packet.
    pub udp_wv: mpsc::Receiver<WorldView>,

    /// Recieves a notification if the TCP connection to the master has failed.
    pub connection_to_master_failed: mpsc::Receiver<bool>,

    /// Recieves elevator containers recieved from slaves on TCP.
    pub container: mpsc::Receiver<ElevatorContainer>,

    /// Recieves requests to remove a container by ID.
    pub remove_container: mpsc::Receiver<u8>,
    /// Recieves TCP container messages that have been transmitted.
    pub sent_tcp_container: mpsc::Receiver<ElevatorContainer>,

    /// Recieves delegated tasks from the manager
    pub delegated_tasks: mpsc::Receiver<HashMap<u8, Vec<[bool; 2]>>>,

    /// Recieves ElevatorContainer from the local elevator handler
    pub elevator_states: mpsc::Receiver<ElevatorContainer>,

    /// Recieves new worldview after reconnecting to the network
    pub new_wv_after_offline: mpsc::Receiver<WorldView>,

    /// Additional buffer channels which can be used if necessary
    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.
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_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,
            },
        }
    }
}
```

Innhald frå Rust-filer

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

```rust
//! This private module provides helper functions for modifying and merging `WorldView` instances
//! within the distributed elevator control system.
//!
//! It is only accessible from the `local_network` module and is responsible for:
//! - Merging local elevator state into the global worldview (via UDP, TCP, or reconnection).
//! - Cleaning up after disconnections (e.g., removing other elevators from the view).
//! - Updating hall/cab requests and elevator statuses after communication.
//! - Resolving conflicts between competing `WorldView` instances during re-entry or master election.
//!
//! ## Purpose
//! In a distributed elevator network, each elevator must maintain an up-to-date view of all other
//! elevators and tasks. This module supports that goal by:
//! - Integrating state from slaves into the master's worldview.
//! - Ensuring task handover and cleanup is handled consistently.
//! - Managing elevator status transitions between connected and offline states.
//!
//! ## Scope and Access
//! This module is private by design and tightly coupled with `local_network`. It is not intended
//! for external access or direct use by high-level logic such as elevator motion or button handling.
//!
//! ## Common Usage Scenarios
//! - Receiving a `WorldView` over UDP during initial network discovery.
//! - Integrating a serialized `ElevatorContainer` received via TCP from a slave.
//! - Cleaning the worldview when a node disconnects or goes offline.
//! - Distributing or clearing tasks after successful transmission.
//!
//! ## Design Notes
//! The merging logic assumes known roles (e.g., master or slave) and behaves accordingly.
//! For instance, a newly elected master may overwrite or discard conflicting data to maintain
//! system consistency.

use crate::world_view::{self, Dirn, ElevatorBehaviour, ElevatorContainer, WorldView};
use crate::{config, print};
use crate::network;

use std::collections::HashMap;
use std::sync::Mutex;
use std::time::{Duration, Instant};




/// 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_from_udp(my_wv: &mut WorldView, master_wv: &mut WorldView) -> bool {
    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.elevator_containers[i_org];
        let master_view = &mut master_wv.elevator_containers[i_new];



        // Synchronize elevator 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;
        master_view.unsent_hall_request = my_view.unsent_hall_request.clone();
        master_view.cab_requests = my_view.cab_requests.clone();


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

    *my_wv = master_wv.clone();
    true
}

/// ### '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.
///
```

```rust
/// ## 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 WorldView) -> bool {
    wv.elevator_containers.retain(|elevator| elevator.elevator_id == network::read_self_id());
    wv.set_num_elev(wv.elevator_containers.len() as u8);
    wv.master_id = network::read_self_id();
    //TODO hent ut self index istedenfor 0 indeks
    wv.hall_request = merge_hall_requests(&wv.hall_request, &wv.elevator_containers[0].tasks);
    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 WorldView, container: &ElevatorContainer) -> bool {

    // If the slave does not exist, add it as-is
    if None == wv.elevator_containers.iter().position(|x| x.elevator_id == container.elevator_id) {
        wv.add_elev(container.clone());
    }

    let self_idx = world_view::get_index_to_container(container.elevator_id, &wv);

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

```
    // Add the slave's sent hall_requests to worldview's hall_requests
    for (row1, row2) in wv.hall_request.iter_mut().zip(container.unsent_hall_request.iter()) {
        for (i, (val1, (j, val2))) in row1.iter_mut().zip(row2.iter().enumerate()).enumerate() {
            if !*val1 && *val2 {
                *val1 = true;

            }
        }
    }




    // Add slaves unfinished tasks to hall_requests
     if wv.elevator_containers[i].behaviour != ElevatorBehaviour::ObstructionError || wv.elevator_containers[i].behaviour
!= ElevatorBehaviour::TravelError {
        wv.hall_request = merge_hall_requests(&wv.hall_request, &wv.elevator_containers[i].tasks);
    }

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

    //Update statuses
    wv.elevator_containers[i].cab_requests = container.cab_requests.clone();
    wv.elevator_containers[i].elevator_id = container.elevator_id;
    wv.elevator_containers[i].last_floor_sensor = container.last_floor_sensor;
    wv.elevator_containers[i].num_floors = container.num_floors;
    wv.elevator_containers[i].obstruction = container.obstruction;
    wv.elevator_containers[i].dirn = container.dirn;
    wv.elevator_containers[i].behaviour = container.behaviour;
    wv.elevator_containers[i].last_behaviour = container.last_behaviour;



    //Remove taken hall_requests
    for (idx, [up, down]) in wv.hall_request.iter_mut().enumerate() {
            if (wv.elevator_containers[i].behaviour == ElevatorBehaviour::DoorOpen) &&
(wv.elevator_containers[i].last_floor_sensor == (idx as u8)) {
            let floor = wv.elevator_containers[i].last_floor_sensor as usize;
            let dirn = match wv.elevator_containers[i].dirn {
                Dirn::Down => Some(1),
                Dirn::Up => Some(0),
                Dirn::Stop => None,
            };

            if wv.elevator_containers[i].last_behaviour != ElevatorBehaviour::DoorOpen {
                update_hall_instants(floor, Some(0));
                update_hall_instants(floor, Some(1));
            }

             if wv.elevator_containers[i].dirn == Dirn::Up  && time_since_hall_instants(floor, dirn) > Duration::from_secs(3)
{
```

```
                    *up = false;
                } else if wv.elevator_containers[i].dirn == Dirn::Down && time_since_hall_instants(floor, dirn) > Duration::from_secs(3) {
                    *down = false;
                }
            }
        }

        // Back up the cab requests
        update_cab_request_backup(&mut wv.cab_requests_backup, wv.elevator_containers[i].clone());

        return true;
    } else {
         // If this is printed, the slave does not exist in the worldview. This is theoretically impossible, as the slave is added to the worldview just before this if it does not already exist.
        print::cosmic_err("The elevator does not exist join_wv_from_tcp_conatiner()".to_string());
        return false;
    }
}

use std::sync::LazyLock;
static HALL_INSTANTS: LazyLock<Mutex<[[Instant; 2]; 4]>> = LazyLock::new(|| {
    Mutex::new(std::array::from_fn(|_| {
        std::array::from_fn(|_| Instant::now())
    }))
});

fn update_hall_instants(floor: usize, direction: Option<usize>) {
    if let Some(dirn) = direction {
        let mut lock = HALL_INSTANTS.lock().unwrap();
        lock[floor][dirn] = Instant::now();
    }
}

fn time_since_hall_instants(floor: usize, direction: Option<usize>) -> std::time::Duration {
    if let Some(dirn) = direction {
        let lock = HALL_INSTANTS.lock().unwrap();
        return lock[floor][dirn].elapsed()
    }
    return Instant::now().elapsed();
}


/// ### 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 WorldView, id: u8) -> bool {
    wv.remove_elev(id);
    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 WorldView, tcp_container: ElevatorContainer) -> bool {
    let self_idx = world_view::get_index_to_container(network::read_self_id() , &wv);

    if let Some(i) = self_idx {
        /*_____ Remove sent Hall request _____ */

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

```
                }
            }
        }
        return true;
    } else {
        // If this is printed, you do not exist in your worldview
        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 WorldView, map: HashMap<u8, Vec<[bool; 2]>>) -> bool {
    for elev in wv.elevator_containers.iter_mut() {
        if let Some(tasks) = map.get(&elev.elevator_id) {
            elev.tasks = tasks.clone();
        }
    }

    true
}


/// Updates states to the elevator in wv with same ID as container
pub fn update_elev_states(wv: &mut WorldView, container: ElevatorContainer) -> bool {
    let idx = world_view::get_index_to_container(container.elevator_id, wv);

    if let Some(i) = idx {
        wv.elevator_containers[i].cab_requests = container.cab_requests;
        wv.elevator_containers[i].dirn = container.dirn;
        wv.elevator_containers[i].obstruction = container.obstruction;
        wv.elevator_containers[i].behaviour = container.behaviour;
        wv.elevator_containers[i].last_behaviour = container.last_behaviour;
        wv.elevator_containers[i].last_floor_sensor = container.last_floor_sensor;
        wv.elevator_containers[i].unsent_hall_request = container.unsent_hall_request;
    }
    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 WorldView, read_wv: &mut WorldView) {
    /* If you become the new master on the system */
    if my_wv.master_id < read_wv.master_id {
        read_wv.hall_request = merge_hall_requests(&read_wv.hall_request, &my_wv.hall_request);
        read_wv.master_id = my_wv.master_id;
        let my_wv_elevs: Vec<ElevatorContainer> = my_wv.elevator_containers.clone();

        /* Map the IDs in the networks worldview */
        let existing_ids: std::collections::HashSet<u8> = read_wv
            .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.elevator_containers.push(elev);
            }
        }

    } else {
        read_wv.hall_request = merge_hall_requests(&read_wv.hall_request, &my_wv.hall_request);
    }

    *my_wv = read_wv.clone();
}

/// 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();
    merged_hall_req
        .iter_mut()
        .zip(hall_req_2)
        .for_each(|(read, my)| {
            read[0] |= my[0];
            read[1] |= my[1];
        });

    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
}
```