Innhald frå Rust-filer

Fil: 975af7d1/snapshot/code/src/main.rs

```
use clap::Parser;
use crossbeam_channel as cbc;
use driver_rust::elevio::elev::Elevator;
use env_logger;
use log::{error, LevelFilter};
use petname::Generator;
use std::thread::spawn;

// Local modules
pub mod config;
pub mod distribute_orders;
pub mod elevator;
pub mod message;
pub mod networking;
pub mod order;
pub mod types;
pub mod single_elevator {
    pub mod elevator;
    pub mod fsm;
    // pub mod main;
    pub mod elevator_controller;
    pub mod requests;
    pub mod timer;
}

#[derive(Debug, Parser)]
struct Args {
    #[arg(long, short, default_value_t = 15657)]
    server_port: u16,

    #[arg(long, short, default_value_t = 19738)]
    peer_port: u16,

    #[arg(long, short, default_value_t = 19735)]
    message_port: u16,
}

fn main() {
    // Sleep for 1 second to allow the server to start
    // std::thread::sleep(std::time::Duration::from_secs(1));

    std::env::set_var("RUST_BACKTRACE", "1");

    env_logger::Builder::new()
        .filter_level(LevelFilter::Trace)
        .init();

    let cli_args = Args::parse();

    let config = config::Config::load().expect("Failed to read config file");
```

```
    let elevio_driver = match Elevator::init(
        format!("localhost:{}", cli_args.server_port).as_str(),
        config.number_of_floors,
    ) {
        Ok(driver) => driver,
        Err(e) => {
            error!(
                "Error initializing elevio driver: {}. Did you remember to start the server first?",
                e
            );
            return;
        }
    };

    let alliterations_generator = petname::Alliterations::default();
    let unique_name = alliterations_generator
        .generate_one(3, "-")
        .expect("Failed to generate unique name with alliterations");

    // Initialize network
    let (command_channel_tx, command_channel_rx) = cbc::unbounded::<types::Orders>();
    let network = networking::Network::new(
        config.clone(),
        cli_args.peer_port,
        cli_args.message_port,
        unique_name.clone(),
        command_channel_tx,
    );

    // Start controller for single elevator
    let network_name = network.network_node_name.clone();
    let network_send_tx = network.data_send_tx.clone();
    spawn(move || {
        single_elevator::elevator_controller::run_controller(
            config.clone(),
            elevio_driver,
            network_name.clone(),
            network_send_tx.clone(),
            command_channel_rx,
        )
    });

    network.start_listening();
}
```

Innhald frå Rust-filer

Fil: 975af7d1/snapshot/code/src/config.rs

```
use log::info;
use serde::{Deserialize, Serialize};
// use std::fs::File;
use std::fs::OpenOptions;
use std::io::BufReader;

/*
================================
 Section for defining constants
================================
*/
// This should really be implemented using a struct of available button presses, as this won't (shouldn't) be able to change
pub const NUM_BUTTONS: u8 = 3;
/*
================================
        End of section
================================
*/

#[derive(Debug, Clone, Serialize, Deserialize)]
pub enum ClearRequestVariant {
    All,
    InDir,
}

#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct Config {
    pub number_of_elevators: u8,
    pub number_of_floors: u8,
    pub polling_interval_ms: u64, // u64 since this is what Duration::from_millis() expects
    pub clear_request_variant: ClearRequestVariant,
    pub door_open_duration_seconds: f64,
    pub simulation_travel_duration_seconds: f64,
}

/// Load configuration from file config.json
/// If the file does not exist, it will be created with default values
impl Config {
    pub fn load() -> Result<Self, Box<dyn std::error::Error>> {
        let path = "config.json";
        let file = OpenOptions::new()
            .read(true)
            .write(true)
            .open(path)
            .unwrap_or_else(|e| {
                panic!("Failed to open config-file: {e}");
            });

        let reader = BufReader::new(file);
```

```
        let config: Config = serde_json::from_reader(reader)?;
        Ok(config)
    }

    pub fn print(&self) {
        info!("====================== CONFIGURATION ======================");
        info!("Number of elevators: {}", self.number_of_elevators);
        info!("Number of floors: {}", self.number_of_floors);
        info!(
            "Polling interval: {} milliseconds",
            self.polling_interval_ms
        );
        info!("Clear request variant: {:?}", self.clear_request_variant);
        info!(
            "Door open duration: {} seconds",
            self.door_open_duration_seconds
        );
        info!(
            "Simulation travel duration: {} seconds",
            self.simulation_travel_duration_seconds
        );
        info!("===========================================================");
    }
}
```

Innhald frå Rust-filer

Fil: 975af7d1/snapshot/code/src/order.rs

```
use crate::types::Direction;

#[derive(Debug, Clone, PartialEq)]
pub struct Order {
    pub floor: u8,
}

#[derive(Debug, Clone)]
pub struct HallOrders {
    pub up: Vec<Order>,
    pub down: Vec<Order>,
}

impl HallOrders {
    pub fn new() -> HallOrders {
        HallOrders {
            up: Vec::new(),
            down: Vec::new(),
        }
    }

    pub fn add_order(&mut self, dir: Direction, floor: u8) {
        let order = Order { floor };
        match dir {
            Direction::Up => self.up.push(order),
            Direction::Down => self.down.push(order),
            _ => (),
        }
    }
}
```

Innhald frå Rust-filer

Fil: 975af7d1/snapshot/code/src/distribute_orders.rs

```
use crate::config::Config;
use crate::elevator::Elevator;
use crate::order::HallOrders;

use log::{debug, error};
use serde_json::{json /*Value*/};
use std::collections::HashMap;
use std::error::Error;
use std::process::Command;

// #[derive(Debug)]
// pub struct DistributionEntry {
//     pub elevator: Elevator,
//     pub requests: HallOrders,
// }
pub type Distribution = HashMap<String, HallOrders>;

pub fn distribute_orders(
    config: &Config,
    elevators: Vec<Elevator>,
    requests: HallOrders,
) -> Result<Distribution, Box<dyn Error>> {
    // Determine executable based on OS
    let os = std::env::consts::OS;
    let executable = match os {
        "linux" => "hall_request_assigner_linux",
        "macos" => "hall_request_assigner_macos",
        _ => {
            let err_msg = format!("Unsupported OS: {os}");
            // error!("{err_msg}");
            return Err(err_msg.into());
        }
    };

    // Fill hall requests array with existing orders
    const UP_INDEX: usize = 0;
    const DOWN_INDEX: usize = 1;
    let mut hall_requests = vec![[false, false]; config.number_of_floors as usize];
    for request in &requests.up {
        hall_requests[request.floor as usize][UP_INDEX] = true;
    }
    for request in &requests.down {
        hall_requests[request.floor as usize][DOWN_INDEX] = true;
    }
    debug!("Hall requests: {:?}", hall_requests);

    // Create JSON object for each elevator
    let mut elevator_states = HashMap::new();
    for elevator in &elevators {
        let elevator_name = &elevator.network_node_name;
```

```
        let current_floor = match elevator.current_floor {
            Some(floor) => floor,
            None => {
                let err_msg = format!("Current floor of {elevator_name} is None");
                // error!("{err_msg}");
                return Err(err_msg.into());
            }
        };

        let direction = match elevator.direction {
            Some(dir) => dir,
            None => {
                let err_msg = format!("Direction of {elevator_name} is None");
                // error!("{err_msg}");
                return Err(err_msg.into());
            }
        };

        let mut cab_requests = vec![false; config.number_of_floors as usize];
        for request in &elevator.cab_orders {
            cab_requests[request.floor as usize] = true;
        }

        let state = json!({
            "behaviour": elevator.behaviour.to_string(),
            "floor": current_floor,
            "direction": direction.to_string(),
            "cabRequests": cab_requests
        });

        elevator_states.insert(elevator.network_node_name.clone(), state);
    }
    debug!("Elevator states: {:?}", elevator_states);

    // Construct JSON input
    let input_json = json!({
        "hallRequests": hall_requests,
        "states": elevator_states
    });

    // Serialize to a string
    let input_json_string = serde_json::to_string_pretty(&input_json).map_err(|e| {
        let err_msg = format!("Failed to serialize input JSON: {e}");
        // error!("{err_msg}");
        err_msg
    })?;
    debug!("Input JSON: {}", input_json_string);

    // Call external process
    let output = Command::new(format!("src/binaries/{}", executable))
        // .arg("--includeCab") // Not necessary, since the cab orders have been included in the simulation, and can only be cleared by the elevator itself anyway
```

```
    .arg("--input")
    .arg(&input_json_string)
    .output()
    .map_err(|e| {
        let err_msg = format!("Failed to execute external command: {e}");
        // error!("{err_msg}");
        err_msg
    })?;

if !output.status.success() {
    let err_msg = format!("External process failed with status: {:?}", output.status);
    // error!("{err_msg}");
    return Err(err_msg.into());
}
debug!("Command executed successfully");

// Convert output to string
let output_string = String::from_utf8(output.stdout).map_err(|e| {
    let err_msg = format!("Failed to convert output to UTF-8: {e}");
    // error!("{err_msg}");
    err_msg
})?;
debug!("Output string: {output_string}");

// Parse output JSON
let output_json: serde_json::Value = serde_json::from_str(&output_string).map_err(|e| {
    let err_msg = format!("Failed to parse output JSON: {e}");
    // error!("{err_msg}");
    err_msg
})?;
debug!("Output JSON: {:#?}", output_json);

let output_obj = match output_json.as_object() {
    Some(obj) => obj,
    None => {
        let err_msg = format!("Expected JSON object, but got: {:#?}", output_json);
        // error!("{err_msg}");
        return Err(err_msg.into());
    }
};

// Convert JSON response to `Distribution`
let mut order_distribution: Distribution = HashMap::new();
for (elevator_name, entry) in output_obj {
    let mut requests = HallOrders {
        up: Vec::new(),
        down: Vec::new(),
    };
    let floors = entry.as_array().ok_or("Expected entry to be an array")?;
    for (floor, floor_requests) in floors.iter().enumerate() {
        // Ensure `floor_requests` is also a valid JSON array
        let directions = floor_requests.as_array().ok_or("Expected floor_requests to be an array")?;
```

```
        for (dir, request) in directions.iter().enumerate() {
            let is_requested = request.as_bool().ok_or("Expected request to be a boolean")?;
            if is_requested {
                let direction = match dir {
                    0 => crate::types::Direction::Up,
                    1 => crate::types::Direction::Down,
                    _ => {
                        let err_msg = format!("Invalid direction index: {dir}");
                        error!("{err_msg}");
                        return Err(err_msg.into());
                    }
                };
                requests.add_order(direction, floor as u8);
            }
        }
    }
    order_distribution.insert(elevator_name.to_string(), requests);
  }

  debug!("Final distribution: {:#?}", order_distribution);
  Ok(order_distribution)
}
```

Innhald frå Rust-filer

Fil: 975af7d1/snapshot/code/src/elevator.rs

```
use crate::order::Order;
use crate::single_elevator::elevator as single_elevator;
use crate::types;

#[derive(Debug, Clone, PartialEq)]
pub struct Elevator {
    pub network_node_name: String,
    // pub network_node_id: types::NetworkNodeId, // If the names collide too often, replace with network_node_id
    pub current_floor: Option<u8>,
    pub behaviour: single_elevator::Behaviour,
    pub direction: Option<types::Direction>,
    pub cab_orders: Vec<Order>,
}

impl Elevator {
    pub fn new(network_node_name: String) -> Self {
        Self {
            network_node_name: network_node_name,
            current_floor: None,
            behaviour: single_elevator::Behaviour::Idle,
            direction: None,
            cab_orders: Vec::new(),
        }
    }
}
```

Innhald frå Rust-filer

Fil: 975af7d1/snapshot/code/src/message.rs

```
use serde;

use crate::single_elevator::elevator::Behaviour;
use crate::types;

#[derive(serde::Serialize, serde::Deserialize, Debug, Clone)]
pub struct HallOrderMessage {
    pub floor: types::Floor,
    pub direction: types::Direction,
}

/// Send a message that an elevator has received a cab order
#[derive(serde::Serialize, serde::Deserialize, Debug, Clone)]
pub struct CabOrderMessage {
    // pub elevator_id: types::ElevatorId,
    pub floor: types::Floor,
}

#[derive(serde::Serialize, serde::Deserialize, Debug, Clone)]
pub struct ElevatorEventMessage {
    // pub elevator_id: types::ElevatorId,
    pub behaviour: Behaviour,
    pub floor: u8,
    pub direction: types::Direction,
}

#[derive(serde::Serialize, serde::Deserialize, Debug, Clone)]
pub struct DataMessage {
    // join all other structs
    pub sender_node_name: String,
    pub message_id: types::MessageId,
    pub hall_order_message: Option<HallOrderMessage>,
    pub cab_order_message: Option<CabOrderMessage>,
    pub elevator_event_message: Option<ElevatorEventMessage>,
}

pub enum MessageType {
    HallOrder(HallOrderMessage),
    CabOrder(CabOrderMessage),
    ElevatorEventMessage(ElevatorEventMessage),
    Unknown,
}

pub trait Message {
    fn to_data_message(self, sender_node_name: &String) -> DataMessage;
}

macro_rules! impl_message {
    ($msg_type:ty, $field:ident) => {
        impl Message for $msg_type {
```

```
        fn to_data_message(self, sender_node_name: &String) -> DataMessage {
            let mut data_message = DataMessage {
                message_id: uuid::Uuid::new_v4().as_u128(),
                sender_node_name: sender_node_name.to_string(),
                hall_order_message: None,
                cab_order_message: None,
                elevator_event_message: None,
            };
            data_message.$field = Some(self);
            data_message
        }
    }
  };
}

impl_message!(HallOrderMessage, hall_order_message);
impl_message!(CabOrderMessage, cab_order_message);
impl_message!(ElevatorEventMessage, elevator_event_message);
```

Innhald frå Rust-filer

Fil: 975af7d1/snapshot/code/src/types.rs

```
use crate::config;
use crate::order::HallOrders;
use serde;

pub type Floor = u8;
pub type ElevatorId = String;
pub type NetworkNodeId = u128;
pub type MessageId = u128;

#[rustfmt::skip] // Prevent rustfmt from reordering the enum variants
#[derive(PartialEq, Copy, Clone, serde::Serialize, serde::Deserialize, Debug)]
pub enum Direction {
    Up   =  1,
    Down = -1,
    Stop =  0,
}

#[derive(Debug)]
pub struct Orders(Vec<[bool; config::NUM_BUTTONS as usize]>); // Struct in order to have default new() function
impl Orders {
    pub fn new(config: &config::Config) -> Self {
        Orders(vec![
            [false; config::NUM_BUTTONS as usize];
            config.number_of_floors as usize
        ])
    }

    // Convert from HallOrders to Orders
    pub fn from_hall_orders(hall_orders: &HallOrders, config: &config::Config) -> Self {
        let mut orders = Self::new(config);

        // Process up orders
        for order in &hall_orders.up {
            if order.floor < config.number_of_floors {
                orders[order.floor as usize][0] = true; // 0 index for UP
            }
        }

        // Process down orders
        for order in &hall_orders.down {
            if order.floor < config.number_of_floors {
                orders[order.floor as usize][1] = true; // 1 index for DOWN
            }
        }

        orders
    }
}

impl std::ops::Index<usize> for Orders {
```

```
    type Output = [bool; config::NUM_BUTTONS as usize];

    fn index(&self, index: usize) -> &Self::Output {
        &self.0[index]
    }
}

impl std::ops::IndexMut<usize> for Orders {
    fn index_mut(&mut self, index: usize) -> &mut Self::Output {
        &mut self.0[index]
    }
}
```

Innhald frå Rust-filer

Fil: 975af7d1/snapshot/code/src/networking.rs

```
// use std::env;
// use std::net;
use log::{error, info, warn};
use std::process;
use std::thread::*;
use std::time::Duration;
// use serde::de;
// use uuid;

use crossbeam_channel as cbc;
use network_rust::udpnet;

use crate::config::Config;
use crate::distribute_orders;
use crate::elevator;
use crate::message::{self, Message};
use crate::order;
use crate::types;
use crate::types::Orders;

pub struct Network {
    config: Config,

    pub network_node_name: String,

    local_state: elevator::Elevator,
    peer_states: Vec<elevator::Elevator>,
    hall_orders: order::HallOrders,

    pub elevator_commands_tx: cbc::Sender<types::Orders>,

    pub peer_sender_tx: cbc::Sender<bool>,
    pub peer_receiver_rx: cbc::Receiver<udpnet::peers::PeerUpdate>,

    pub data_receiver_rx: cbc::Receiver<message::DataMessage>,
    pub data_send_tx: cbc::Sender<message::DataMessage>,
}

impl Network {
    /**
     * Start the UDP socket for peer discovery.
     */
    fn start_discovery_transmit(peer_port: u16, unique_name: String) -> cbc::Sender<bool> {
        let (peer_tx_enable_tx, peer_tx_enable_rx) = cbc::unbounded::<bool>();
        {
            // let id = process::id().to_string();
            spawn(move || {
                let result = udpnet::peers::tx(peer_port, unique_name, peer_tx_enable_rx);
                if result.is_err() {
                    error!(
```

```
                    "Failed to start peer discovery transmit: {}",
                    result.err().unwrap()
                );
                std::thread::sleep(Duration::from_secs(1));
                  process::exit(1); // crash program if creating the socket fails (`peers:tx` will always block if the initialization
succeeds)
            }
        })
    };
    return peer_tx_enable_tx;
}

/**
 * Create receiver for peer discovery information.
 */
fn start_discovery_receive(peer_port: u16) -> cbc::Receiver<udpnet::peers::PeerUpdate> {
    let (peer_update_tx, peer_update_rx) = cbc::unbounded::<udpnet::peers::PeerUpdate>();
    {
        spawn(move || {
            let result = udpnet::peers::rx(peer_port, peer_update_tx);
            if result.is_err() {
                error!(
                    "Failed to start peer discovery receive: {}",
                    result.err().unwrap()
                );
                std::thread::sleep(Duration::from_secs(1));
                  process::exit(1); // crash program if creating the socket fails (`peers:rx` will always block if the initialization
succeeds)
            }
        });
    }
    return peer_update_rx;
}

fn initate_channel_for_recieving_data(msg_port: u16) -> cbc::Receiver<message::DataMessage> {
    let (data_receiver_tx, data_receiver_rx) = cbc::unbounded::<message::DataMessage>();
    {
        spawn(move || {
            let result = udpnet::bcast::rx(msg_port, data_receiver_tx);
            if result.is_err() {
                error!("Failed to start data receive: {}", result.err().unwrap());
                std::thread::sleep(Duration::from_secs(1));
                  process::exit(1); // crash program if creating the socket fails (`bcast:rx` will always block if the initialization
succeeds)
            }
        });
    }
    return data_receiver_rx;
}

fn initiate_channel_for_sending_data(msg_port: u16) -> cbc::Sender<message::DataMessage> {
    let (data_send_tx, data_send_rx) = cbc::unbounded::<message::DataMessage>();
```

```
        {
            spawn(move || {
                let result = udpnet::bcast::tx(msg_port, data_send_rx);
                if result.is_err() {
                    error!("Failed to start data send: {}", result.err().unwrap());
                    std::thread::sleep(Duration::from_secs(1));
                      process::exit(1); // crash program if creating the socket fails (`bcast:tx` will always block if the initialization succeeds)
                }
            });
        }
        return data_send_tx;
    }

    /**
     * Create a new network and begin peer discovery.
     * Also sets up the UDP socket for receiving and sending data messages.
     */
    pub fn new(
        config: Config,
        peer_port: u16,
        message_port: u16,
        unique_name: String,
        elevator_commands_tx: cbc::Sender<types::Orders>,
    ) -> Network {
        let network = Network {
            config,

            network_node_name: unique_name.clone(),

            local_state: elevator::Elevator::new(unique_name.clone()),
            peer_states: Vec::new(),
            hall_orders: order::HallOrders::new(),

            elevator_commands_tx: elevator_commands_tx,
            peer_sender_tx: Self::start_discovery_transmit(peer_port, unique_name),
            peer_receiver_rx: Self::start_discovery_receive(peer_port),
            data_receiver_rx: Self::initate_channel_for_recieving_data(message_port),
            data_send_tx: Self::initiate_channel_for_sending_data(message_port),
        };

        return network;
    }

    pub fn start_listening(mut self) {
        loop {
            cbc::select! {
                recv(self.peer_receiver_rx) -> received => {
                    let update = match received {
                        Ok(update) => update,
                        Err(e) => {
                            error!("Error receiving peer update: {e}");
```

```
                continue;
            }
        };

        // Check for new peers
        if let Some(peer_name) = &update.new {
            let is_local = peer_name == &self.network_node_name;
            let peer = self.find_peer(&peer_name.to_string());
            if !is_local && peer.is_none() {
                self.new_peer_procedure(peer_name);
            }
        }

        // Check for lost peers
        if !update.lost.is_empty() {
            for lost_peer in &update.lost {
                if let Some(index) = self.peer_states.iter().position(|e| e.network_node_name == lost_peer.to_string()) {
// Maybe move to a function `find_peer_index`
                    self.peer_states.remove(index);
                }
                info!("Lost elevator: {:#?}", lost_peer);
            }
        }
    }
    recv(self.data_receiver_rx) -> received => {
        let message = match received {
            Ok(message) => message,
            Err(e) => {
                error!("Error receiving data message: {e}");
                continue;
            }
        };
        match Self::infer_message_type(&message) {
            message::MessageType::HallOrder(hall_order_message) => {
                self.process_hall_order(hall_order_message, message.sender_node_name);
            }
            message::MessageType::CabOrder(cab_order_message) => {
                self.process_cab_order(cab_order_message, message.sender_node_name);
            }
            message::MessageType::ElevatorEventMessage(elevator_event_message) => {
                self.process_event(elevator_event_message, message.sender_node_name);
            }
            message::MessageType::Unknown => {
                warn!("Unknown message type received.");
            }
        }
    }

    // Default
    default(Duration::from_millis(500)) => {
        // debug!("Controller default");
    }
```

```
        }
    }
}

fn infer_message_type(message: &message::DataMessage) -> message::MessageType {
    if let Some(hall_order) = &message.hall_order_message {
        return message::MessageType::HallOrder(hall_order.clone());
    }
    if let Some(cab_order) = &message.cab_order_message {
        return message::MessageType::CabOrder(cab_order.clone());
    }
    if let Some(elevator_event) = &message.elevator_event_message {
        return message::MessageType::ElevatorEventMessage(elevator_event.clone());
    }
    message::MessageType::Unknown
}

fn process_hall_order(
    &mut self,
    hall_order: message::HallOrderMessage,
    sender_node_name: String,
) {
    info!(
        "Hall order received from {sender_node_name}: {:#?}",
        hall_order
    );

    // Run order distribution logic
    self.hall_orders
        .add_order(hall_order.direction, hall_order.floor);

    // Combine all peers including local state
    let all_elevators = {
        let mut peers = self.peer_states.to_vec();
        peers.push(self.local_state.clone());
        peers
    };

    // Calculate new order distribution
    let new_order_distribution = distribute_orders::distribute_orders(
        &self.config,
        all_elevators,
        self.hall_orders.clone(),
    );

    // Handle new order distribution
    let new_order_distribution = match new_order_distribution {
        Ok(distribution) => {
            info!("New order distribution: {:#?}", distribution);
            distribution
        }
        Err(e) => {
```

```
            error!("Failed to distribute orders: {e}");
            return;
        }
    };
    let local_node_name = &self.network_node_name;
    let local_order_distribution = match new_order_distribution.get(&self.network_node_name) {
        Some(local_distribution) => {
            info!(
                "Local distribution ({local_node_name}): {:#?}",
                local_distribution
            );
            local_distribution
        }
        None => {
            error!("Failed to get local distribution.");
            return;
        }
    };

    // Convert from HallOrders to Orders
    let orders = Orders::from_hall_orders(local_order_distribution, &self.config);
    self.elevator_commands_tx.send(orders).unwrap();
}

fn process_cab_order(&self, cab_order: message::CabOrderMessage, sender_node_name: String) {
    info!(
        "Cab order received from {sender_node_name}: {:#?}",
        cab_order
    );

    // Update elevator state
    // let elevator = elevator_states.iter_mut().find(|e| e.network_node_name == sender_node_name);
    // match elevator {
    //     Some(e) => {
    //         info!("Old elevator state: {:#?}", e);

    //         e.cab_orders.push(order::Order::new(cab_order.floor));

    //         info!("New elevator state: {:#?}", e);
    //     },
    //     None => {
    //         error!("Elevator with name {} not found in elevator_states", sender_node_name);
    //     }
    // }
}

fn process_event(&mut self, event: message::ElevatorEventMessage, sender_node_name: String) {
    // Check if the event is for the local elevator
    if self.network_node_name == sender_node_name {
        self.local_state.current_floor = Some(event.floor);
        self.local_state.behaviour = event.behaviour;
        self.local_state.direction = Some(event.direction);
```

```
        info!("Event - updated local state: {:#?}", self.local_state);
        return;
    }

    // Find or create a peer for this sender
    let peer = match self.find_peer_mut(&sender_node_name) {
        Some(existing_peer) => existing_peer,
        None => {
            info!("Event from unregistered peer {sender_node_name}");
            self.new_peer_procedure(&sender_node_name)
        }
    };

    // Update peer state
    peer.current_floor = Some(event.floor);
    peer.behaviour = event.behaviour;
    peer.direction = Some(event.direction);
    info!("Event - updated peer state: {:#?}", peer);
}

fn find_peer(&self, peer_name: &String) -> Option<&elevator::Elevator> {
    self.peer_states
        .iter()
        .find(|e| e.network_node_name == peer_name.to_string())
}

fn find_peer_mut(&mut self, peer_name: &String) -> Option<&mut elevator::Elevator> {
    self.peer_states
        .iter_mut()
        .find(|e| e.network_node_name == peer_name.to_string())
}

fn new_peer_procedure(&mut self, new_peer: &String) -> &mut elevator::Elevator {
    self.peer_states
        .push(elevator::Elevator::new(new_peer.to_string()));
    let new_elevator = self
        .peer_states
        .last_mut()
        .expect("peer_states should not be empty after push");
    info!("New peer: {:#?}", new_elevator);

    // Checklist - need to:
    // [ ] Send own state to new peer
    // [ ] Determine if new peer rebooted, or reconnected
    //   -> Reboot:    [ ] Send all orders to new peer
    //   -> Reconnect: [ ] Synchronize orders with new peer
    // [ ] Update order distribution

    // Publish own state to new peer (if not currently initializing). This may cause duplicate events to be received.
    if let (Some(current_floor), Some(direction)) =
        (self.local_state.current_floor, self.local_state.direction)
    {
```

```
            let event = message::ElevatorEventMessage {
                behaviour: self.local_state.behaviour,
                floor: current_floor,
                direction: direction,
            };

            if let Err(e) = self
                .data_send_tx
                .send(event.to_data_message(&self.network_node_name))
            {
                error!("Failed to send message: {:?}", e);
            }
        }

        new_elevator
    }
}
```

Innhald frå Rust-filer

Fil: 975af7d1/snapshot/code/src/single_elevator/requests.rs

```
// Crates
use crate::config::ClearRequestVariant;
use crate::single_elevator::elevator;
use crate::types::Direction;
use crate::config::NUM_BUTTONS;

type DirectionBehaviourPair = (Direction, elevator::Behaviour);

pub fn above(e: &elevator::State) -> bool {
    let floor = match e.get_floor() {
        Some(f) => f,
        None => return false,
    };
    (floor + 1..e.config.number_of_floors)
        .any(|f| (0..NUM_BUTTONS)
        .any(|c| e.get_request(f, c.into())))
}

pub fn below(e: &elevator::State) -> bool {
    let floor = match e.get_floor() {
        Some(f) => f,
        None => return false,
    };
    (0..floor)
        .any(|f| (0..NUM_BUTTONS)
        .any(|c| e.get_request(f, c.into())))
}

pub fn here(e: &elevator::State) -> bool {
    let floor = match e.get_floor() {
        Some(f) => f,
        None => return false,
    };
    (0..NUM_BUTTONS)
        .any(|c| e.get_request(floor, c.into()))
}

pub fn choose_direction(e: &elevator::State) -> DirectionBehaviourPair {
    match e.direction {
        Direction::Up => {
            if above(e) {
                (Direction::Up, elevator::Behaviour::Moving)
            } else if here(e) {
                (Direction::Down, elevator::Behaviour::DoorOpen)
            } else if below(e) {
                (Direction::Down, elevator::Behaviour::Moving)
            } else {
                (Direction::Stop, elevator::Behaviour::Idle)
            }
        }
```

```
        Direction::Down => {
            if below(e) {
                (Direction::Down, elevator::Behaviour::Moving)
            } else if here(e) {
                (Direction::Up, elevator::Behaviour::DoorOpen)
            } else if above(e) {
                (Direction::Up, elevator::Behaviour::Moving)
            } else {
                (Direction::Stop, elevator::Behaviour::Idle)
            }
        }
        Direction::Stop => {
            if here(e) {
                (Direction::Stop, elevator::Behaviour::DoorOpen)
            } else if above(e) {
                (Direction::Up, elevator::Behaviour::Moving)
            } else if below(e) {
                (Direction::Down, elevator::Behaviour::Moving)
            } else {
                (Direction::Stop, elevator::Behaviour::Idle)
            }
        }
    }
}

pub fn should_stop(e: &elevator::State) -> bool {
    let floor = match e.get_floor() {
        Some(f) => f,
        None => return false,
    };

    match e.direction {
        Direction::Down => {
            (e.get_request(floor, elevator::Button::HallDown))
                || (e.get_request(floor, elevator::Button::Cab))
                || !below(e)
        }
        Direction::Up => {
            (e.get_request(floor, elevator::Button::HallUp))
                || (e.get_request(floor, elevator::Button::Cab))
                || !above(e)
        }
        Direction::Stop => true,
    }
}

pub fn should_clear_immediately(
    e: &elevator::State,
    btn_floor: u8,
    btn_type: elevator::Button,
) -> bool {
    let floor = match e.get_floor() {
```

```
        Some(f) => f,
        None => return false,
    };

    match e.config.clear_request_variant {
        ClearRequestVariant::All => floor == btn_floor,
        ClearRequestVariant::InDir => {
            floor == btn_floor
                && (e.direction == Direction::Stop
                    || btn_type == elevator::Button::Cab
                    || (e.direction == Direction::Up
                        && btn_type == elevator::Button::HallUp)
                    || (e.direction == Direction::Down
                        && btn_type == elevator::Button::HallDown))
        }
    }
}

pub fn clear_at_current_floor(e: &mut elevator::State) -> &mut elevator::State {
    let floor = match e.get_floor() {
        Some(f) => f,
        None => return e,
    };

    match e.config.clear_request_variant {
        ClearRequestVariant::All => {
            (0..NUM_BUTTONS).for_each(|c| e.set_request(floor, c.into(), false));
        }
        ClearRequestVariant::InDir => {
            e.set_request(floor, elevator::Button::Cab, false);
            match e.direction {
                Direction::Up => {
                    if !above(&e) && !e.get_request(floor, elevator::Button::HallUp) {
                        e.set_request(floor, elevator::Button::HallDown, false);
                    }
                    e.set_request(floor, elevator::Button::HallUp, false);
                }
                Direction::Down => {
                    if !below(&e) && !e.get_request(floor, elevator::Button::HallDown) {
                        e.set_request(floor, elevator::Button::HallUp, false);
                    }
                    e.set_request(floor, elevator::Button::HallDown, false);
                }
                Direction::Stop => {
                    e.set_request(floor, elevator::Button::HallUp, false);
                    e.set_request(floor, elevator::Button::HallDown, false);
                }
            }
        }
    }
    return e;
}
```

Innhald frå Rust-filer

Fil: 975af7d1/snapshot/code/src/single_elevator/elevator_controller.rs

```
use core::panic;
use crossbeam_channel as cbc;
use log::{debug, error, info};
use std::thread;
use std::time;

use driver_rust::elevio;

use crate::config::Config;
use crate::message::{self, Message};
use crate::single_elevator::elevator;
use crate::single_elevator::fsm;
use crate::single_elevator::timer;
use crate::types;
use crate::types::Orders;
// use crate::single_elevator::requests;

pub fn run_controller(
    config: Config,
    elevio_driver: elevio::elev::Elevator,
    network_node_name: String,
    network_tx: cbc::Sender<message::DataMessage>,
    command_rx: cbc::Receiver<Orders>,
) {
    let polling_interval = time::Duration::from_millis(config.polling_interval_ms);

    // Call buttons
    let (call_button_tx, call_button_rx) = cbc::unbounded::<elevio::poll::CallButton>();
    {
        let elevio_driver = elevio_driver.clone();
        thread::spawn(move || {
            elevio::poll::call_buttons(elevio_driver, call_button_tx, polling_interval)
        });
    }

    // Floor sensor
    let (floor_sensor_tx, floor_sensor_rx) = cbc::unbounded::<u8>();
    {
        let elevio_driver = elevio_driver.clone();
        thread::spawn(move || {
            elevio::poll::floor_sensor(elevio_driver, floor_sensor_tx, polling_interval)
        });
    }

    // Obstruction
    let (obstruction_tx, obstruction_rx) = cbc::unbounded::<bool>();
    {
        let elevio_driver = elevio_driver.clone();
        thread::spawn(move || {
            elevio::poll::obstruction(elevio_driver, obstruction_tx, polling_interval)
```

```
    });
}

// Timer
let (timer_elev_tx, timer_elev_rx) = cbc::unbounded::<timer::TimerMessage>();
let (timer_time_tx, timer_time_rx) = cbc::unbounded::<timer::TimerMessage>();
{
    let mut timer_instance = timer::Timer::new();
    thread::spawn(move || loop {
        let mut sel = cbc::Select::new();
        sel.recv(&timer_time_rx); // This IS NECESSARY, but why?

        let oper = sel.try_select();
        match oper {
            Err(_) => {
                // Since try_select is non-blocking, this is the default case when no messages are available
                if timer_instance.timed_out() {
                    timer_instance.stop();
                    timer_elev_tx.send(timer::TimerMessage::TimedOut).unwrap();
                }
            }
            Ok(oper) => {
                    let timer_message = oper.recv(&timer_time_rx).unwrap(); // Sometimes get error "unwrap on Err value: RecvError", needs fixing
                match timer_message {
                    timer::TimerMessage::Start(duration) => {
                        timer_instance.start(duration);
                    }
                    timer::TimerMessage::Stop => {
                        timer_instance.stop();
                    }
                    _ => {}
                }
            }
        }
    });
}

// Initialize elevator
let mut elevator_state = elevator::State::new(config, timer_time_tx);
let initial_obstruction = elevio_driver.obstruction();
elevator_state.obstruction = initial_obstruction;
// Check for initial floor
let initial_floor = elevio_driver.floor_sensor();
if initial_floor == None {
    fsm::on_init_between_floors(&elevio_driver, &mut elevator_state);
}

loop {
    cbc::select! {
        // Command from network
        recv(command_rx) -> received => {
```

```
    let requests = match received {
        Ok(requests) => requests,
        Err(e) => {
            error!("Error receiving new requests: {e}");
            continue;
        }
    };

    // Update local array of requests
    debug!("Received new requests: {:#?}", requests);
    elevator_state.set_all_requests(requests);
    fsm::set_all_lights(&elevio_driver, &elevator_state);

    // Do we need to do something? Only if we're IDLE
    if elevator_state.behaviour == elevator::Behaviour::Idle {
        info!("In IDLE, starting new request");
        // let floor = elevator_state.get_floor().unwrap();
        // fsm::on_arrival(&elevio_driver, &mut elevator_state, floor);
        fsm::on_new_order_assignment(&elevio_driver, &mut elevator_state);
    } else {
        info!("Not IDLE, ignoring new request");
    }
},

// Call button
recv(call_button_rx) -> received => {
    let call_button = match received {
        Ok(call_button) => call_button,
        Err(e) => {
            error!("Error receiving peer update: {e}");
            continue;
        }
    };
    let button = match call_button.call {
        0 => elevator::Button::HallUp,
        1 => elevator::Button::HallDown,
        2 => elevator::Button::Cab,
        _ => panic!("Invalid call button"),
    };
    let direction = match button {
        elevator::Button::HallUp => types::Direction::Up,
        elevator::Button::HallDown => types::Direction::Down,
        _ => types::Direction::Up,
    };
    // debug!("{:#?}", call_button);
    // elevio_driver.call_button_light(call_button.floor, call_button.call, true);
    // fsm::on_request_button_press(&elevio_driver, &mut elevator_state, call_button.floor, button);
    match button {
        elevator::Button::HallUp | elevator::Button::HallDown => {
            // Notify of hall order
            let event: message::HallOrderMessage = message::HallOrderMessage {
                floor: call_button.floor,
```

```
                direction: direction,
            };
            network_tx.send(event.to_data_message(&network_node_name)).unwrap();
        },
        elevator::Button::Cab => {
            // Notify of cab order
            let event: message::CabOrderMessage = message::CabOrderMessage {
                floor: call_button.floor,
            };
            network_tx.send(event.to_data_message(&network_node_name)).unwrap();
        },
    };
},

// Floor sensor
recv(floor_sensor_rx) -> received => {
    let floor = match received {
        Ok(floor) => floor,
        Err(e) => {
            error!("Error receiving floor: {e}");
            continue;
        }
    };
    // if elevator_state.get_floor() != Some(floor) {
      //     // TODO: What happens if elevator goes up, then down to same floor without reaching any other floor?
Need to handle so arrival code executes again
    // }
    debug!("Arrival at floor");
    fsm::on_arrival(&elevio_driver, &mut elevator_state, floor);

    // Notify of an event (updated state)
    let event: message::ElevatorEventMessage = message::ElevatorEventMessage {
        behaviour: elevator_state.behaviour,
        floor: elevator_state.get_floor().unwrap(), // Can unwrap be an issue here?
        direction: elevator_state.direction,
    };
    network_tx.send(event.to_data_message(&network_node_name)).unwrap();
},

// Timer
recv(timer_elev_rx) -> received => {
    let timer_message = match received {
        Ok(timer_message) => timer_message,
        Err(e) => {
            error!("Error receiving timer message: {e}");
            continue;
        }
    };
    debug!("Timer message");
    match timer_message {
        timer::TimerMessage::TimedOut => {
            if elevator_state.obstruction {
```

```
                    debug!("Obstruction detected, restarting door timer");
                    elevator_state.start_door_timer();
                } else {
                    debug!("Door timeout");
                    fsm::on_door_timeout(&elevio_driver, &mut elevator_state);

                    // Notify of an event (updated state)
                    let event: message::ElevatorEventMessage = message::ElevatorEventMessage {
                        behaviour: elevator_state.behaviour,
                        floor: elevator_state.get_floor().unwrap(), // Can unwrap be an issue here?
                        direction: elevator_state.direction,
                    };
                    network_tx.send(event.to_data_message(&network_node_name)).unwrap();
                }
            },
            _ => {},
        }
    },

    // Obstruction
    recv(obstruction_rx) -> received => {
        let obstr = match received {
            Ok(obstr) => obstr,
            Err(e) => {
                error!("Error receiving obstruction: {e}");
                continue;
            }
        };
        elevator_state.obstruction = obstr;
    },

    // Default
    default(time::Duration::from_millis(500)) => {
        // debug!("Controller default");
    },
  }
 }
}
```

Innhald frå Rust-filer

Fil: 975af7d1/snapshot/code/src/single_elevator/elevator.rs

```
use crossbeam_channel as cbc;

use crate::config::{self, NUM_BUTTONS};
use crate::single_elevator::timer;
use crate::types::{Direction, Orders};

// Behaviour
#[derive(PartialEq, Copy, Clone, Debug, serde::Serialize, serde::Deserialize)]
pub enum Behaviour {
    Idle,
    DoorOpen,
    Moving,
}
impl Behaviour {
    // Not yet used
    pub fn to_string(&self) -> String {
        match self {
            Behaviour::Idle => "idle".to_string(),
            Behaviour::Moving => "moving".to_string(),
            Behaviour::DoorOpen => "doorOpen".to_string(),
        }
    }
}

impl Direction {
    pub fn to_string(&self) -> String {
        match self {
            Direction::Up => "up".to_string(),
            Direction::Down => "down".to_string(),
            Direction::Stop => "stop".to_string(),
        }
    }
}

// Button
#[derive(PartialEq, Copy, Clone)]
pub enum Button {
    HallUp,
    HallDown,
    Cab,
}
impl Button {
    pub fn to_string(&self) -> String {
        match self {
            Button::HallUp => "HallUp".to_string(),
            Button::HallDown => "HallDown".to_string(),
            Button::Cab => "Cab".to_string(),
        }
    }
}
```

```
impl From<u8> for Button {
    fn from(value: u8) -> Self {
        match value {
            0 => Button::HallUp,
            1 => Button::HallDown,
            2 => Button::Cab,
            _ => panic!("Invalid button value"),
        }
    }
}

// ClearRequestVariant
pub enum ClearRequestVariant {
    All,
    InDir,
}

// State
pub struct State {
    floor: Option<u8>,
    previous_floor: Option<u8>,
    requests: Orders,

    pub direction: Direction,

    // pub door_timer: timer::Timer,
    pub obstruction: bool,
    pub timer_tx: cbc::Sender<timer::TimerMessage>,
    pub behaviour: Behaviour,

    pub config: config::Config,
}
impl State {
    pub fn print(&self) {
        let floor = match self.floor {
            Some(f) => f.to_string(),
            None => "Undefined".to_string(),
        };

        println!("  +--------------------+");
        println!("  |floor = {:<2}          |", floor);
        println!(
            "  |dirn  = {:<12.12}|",
            match self.direction {
                Direction::Down => "Down",
                Direction::Stop => "Stop",
                Direction::Up => "Up",
            }
        );
        println!(
            "  |behav = {:<12.12}|",
            match self.behaviour {
```

```
            Behaviour::Idle => "Idle",
            Behaviour::DoorOpen => "DoorOpen",
            Behaviour::Moving => "Moving",
        }
    );
    println!(
        "  |obstr = {:<12.12}|",
        match self.obstruction {
            true => "yes",
            false => "no",
        }
    );
    println!("  +--------------------+");
    println!("  |  | up  | dn  | cab |");
    for f in (0..self.config.number_of_floors).rev() {
        print!("  | {}", f);
        for btn in 0..NUM_BUTTONS {
            if (f == self.config.number_of_floors - 1 && btn == Button::HallUp as u8)
                || (f == 0 && btn == Button::HallDown as u8)
            {
                print!("|     ");
            } else {
                print!(
                    "|  {}  ",
                    if self.get_request(f, btn.into()) {
                        "#"
                    } else {
                        "-"
                    }
                );
            }
        }
        println!("|");
    }
    println!("  +--------------------+");
}

pub fn new(config: config::Config, timer_tx: cbc::Sender<timer::TimerMessage>) -> Self {
    Self {
        floor: None,
        previous_floor: None,
        direction: Direction::Stop,
        obstruction: false,
        timer_tx,
        requests: Orders::new(&config),
        behaviour: Behaviour::Idle,
        config,
    }
}

pub fn get_request(&self, floor: u8, button: Button) -> bool {
    assert!(
```

```
            floor < self.config.number_of_floors,
            "Floor out of bounds in get_request",
        );
        self.requests[floor as usize][button as usize]
    }

    pub fn set_request(&mut self, floor: u8, button: Button, value: bool) {
        assert!(
            floor < self.config.number_of_floors,
            "Floor out of bounds in set_request",
        );
        self.requests[floor as usize][button as usize] = value;
    }

    pub fn set_all_requests(&mut self, requests: Orders) {
        self.requests = requests;
    }

    pub fn get_floor(&self) -> Option<u8> {
        self.floor
    }
    // pub fn get_previous_floor(&self) -> Option<u8> {
    //     self.previous_floor
    // }
    pub fn set_floor(&mut self, new_floor: u8) {
        self.previous_floor = self.floor;
        self.floor = Some(new_floor);
    }

    pub fn start_door_timer(&self) {
        self.timer_tx
            .send(timer::TimerMessage::Start(
                self.config.door_open_duration_seconds,
            ))
            .unwrap();
    }
    // pub fn stop_door_timer(&self) {
    //     self.timer_tx.send(timer::TimerMessage::Stop).unwrap();
    // }
}
```

Innhald frå Rust-filer

Fil: 975af7d1/snapshot/code/src/single_elevator/fsm.rs

```
use driver_rust::elevio::elev as e;

use crate::config::NUM_BUTTONS;
use crate::single_elevator::elevator;
use crate::single_elevator::requests;
use crate::types::Direction;

pub fn set_all_lights(elevio_driver: &e::Elevator, elevator_state: &elevator::State) {
    for f in 0..elevator_state.config.number_of_floors {
        for c in 0..NUM_BUTTONS {
            let light = elevator_state.get_request(f.try_into().unwrap(), c.try_into().unwrap());
            elevio_driver.call_button_light(f.try_into().unwrap(), c, light);
        }
    }
}

pub fn on_init_between_floors(elevio_driver: &e::Elevator, elevator_state: &mut elevator::State) {
    elevio_driver.motor_direction(e::DIRN_DOWN);
    elevator_state.direction = Direction::Down;
    elevator_state.behaviour = elevator::Behaviour::Moving;
}

pub fn on_new_order_assignment(
    // This is almost a blind copy of fsm::on_request_button_press
    elevio_driver: &e::Elevator,
    mut elevator_state: &mut elevator::State,
    // request_floor: u8,
    // button: elevator::Button,
) {
    match elevator_state.behaviour {
        elevator::Behaviour::Idle => {
            // elevator_state.set_request(request_floor, button, true);
            let direction_behaviour_pair = requests::choose_direction(&elevator_state);
            elevator_state.direction = direction_behaviour_pair.0;
            elevator_state.behaviour = direction_behaviour_pair.1;
            match elevator_state.behaviour {
                elevator::Behaviour::Idle => {}
                elevator::Behaviour::DoorOpen => {
                    elevio_driver.door_light(true);
                    elevator_state.start_door_timer();
                    elevator_state = requests::clear_at_current_floor(elevator_state);
                    // THIS NEEDS TO BE BROADCAST AS WELL
                }
                elevator::Behaviour::Moving => {
                    elevio_driver.motor_direction(elevator_state.direction as u8);
                }
            }
        }
        elevator::Behaviour::DoorOpen => {
            // if requests::should_clear_immediately(&elevator_state, request_floor, button) {
```

```
            //     elevator_state.start_door_timer();
            // } else {
            //     elevator_state.set_request(request_floor, button, true);
            // }
        }
        elevator::Behaviour::Moving => {
            // elevator_state.set_request(request_floor, button, true);
        }
    }

    set_all_lights(elevio_driver, elevator_state);

    // debug!("New state: "\);
    // elevator_state.print();
}

pub fn on_arrival(
    elevio_driver: &e::Elevator,
    mut elevator_state: &mut elevator::State,
    new_floor: u8,
) {
    // debug!("Arrived at floor {}", new_floor);
    // elevator_state.print();

    elevator_state.set_floor(new_floor);
    elevio_driver.floor_indicator(new_floor);

    match elevator_state.behaviour {
        elevator::Behaviour::Moving => {
            if requests::should_stop(&elevator_state) {
                elevio_driver.motor_direction(e::DIRN_STOP);
                elevio_driver.door_light(true);
                elevator_state = requests::clear_at_current_floor(elevator_state);
                elevator_state.start_door_timer();
                set_all_lights(elevio_driver, elevator_state);
                elevator_state.behaviour = elevator::Behaviour::DoorOpen;
            }
        }
        _ => {}
    }

    // debug!("New state: "\);
    // elevator_state.print();
}

pub fn on_door_timeout(elevio_driver: &e::Elevator, mut elevator_state: &mut elevator::State) {
    // debug!("Door timeout");
    // elevator_state.print();

    match elevator_state.behaviour {
        elevator::Behaviour::DoorOpen => {
            let direction_behaviour_pair = requests::choose_direction(&elevator_state);
```

```
        elevator_state.direction = direction_behaviour_pair.0;
        elevator_state.behaviour = direction_behaviour_pair.1;

        match elevator_state.behaviour {
            elevator::Behaviour::Moving | elevator::Behaviour::Idle => {
                elevio_driver.door_light(false);
                elevio_driver.motor_direction(elevator_state.direction as u8);
            }
            elevator::Behaviour::DoorOpen => {
                elevator_state.start_door_timer();
                 elevator_state = requests::clear_at_current_floor(elevator_state); // THIS NEEDS TO BE BROADCAST AS
WELL
                set_all_lights(elevio_driver, elevator_state);
            }
        }
    }
    _ => {}
  }

  // debug!("New state: "\);
  // elevator_state.print();
}
```

Innhald frå Rust-filer

Fil: 975af7d1/snapshot/code/src/single_elevator/timer.rs

```
use std::fmt;
use std::time::SystemTime;

pub fn get_wall_time() -> f64 {
    let now = SystemTime::now()
        .duration_since(SystemTime::UNIX_EPOCH)
        .unwrap();
    now.as_secs() as f64 + now.subsec_micros() as f64 * 0.000001
}

pub struct Timer {
    end_time: f64,
    active: bool,
}
impl Timer {
    pub fn new() -> Self {
        Timer {
            end_time: 0.0,
            active: false,
        }
    }

    pub fn start(&mut self, duration: f64) {
        self.end_time = get_wall_time() + duration;
        self.active = true;
    }

    pub fn stop(&mut self) {
        self.active = false;
    }

    pub fn timed_out(&self) -> bool {
        self.active && get_wall_time() > self.end_time
    }
}
impl fmt::Debug for Timer {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        write!(
            f,
            "Timer {{ end_time: {}, active: {} }}",
            self.end_time, self.active
        )
    }
}

pub enum TimerMessage {
    Start(f64),
    Stop,
    TimedOut,
}
```