Innhald frå Rust-filer

Fil: dbca849d/snapshot-[REDACTED]/main/src/main.rs

```
use core::time::Duration;
use std::thread::spawn;

use crossbeam_channel as cbc;
use driver_rust::elevio;
use driver_rust::elevio::elev as e;
use log::info;

mod messages;
mod manager;
mod controller;
mod sender;
mod receiver;
mod alarm;
mod lights;
mod fsm;
mod config;
use std::env;

fn main() {

    let args: Vec<String> = env::args().collect();

    let mut id: Option<u8> = None;

    let mut iter = args.iter();
    while let Some(arg) = iter.next() {
        if arg == "--id" {
            if let Some(value) = iter.next() {
                id = value.parse().ok();
            }
        }
    }

    let id = match id {
        Some(id) => id,
        _ => {
            0
        }
    };
    info!("Running with ID {}", id);
    env_logger::init();
    info!("Booting application.");
    // create channels
    info!("Creating channels.");
    let (manager_tx, manager_rx) = cbc::unbounded::<messages::Manager>();
    let (controller_tx, controller_rx) = cbc::unbounded::<messages::Controller>();
    let (lights_tx, lights_rx) = cbc::unbounded::<messages::Controller>();
    let (sender_tx, sender_rx) = cbc::unbounded::<messages::Manager>();
    let (alarm_tx, alarm_rx) = cbc::unbounded::<u8>();
```

```
let (call_button_tx, call_button_rx) = cbc::unbounded::<elevio::poll::CallButton>();

// create elevator_connection object
let elev_num_floors = 4;
// use this if you want to run in a docker container
let default_port = "15657".to_string();
let port = env::var("ELEVATOR_PORT").unwrap_or(default_port);
let address = format!("host.docker.internal:{}", port);
// let address = format!("127.0.0.1:15657");

let elevator_connection = e::Elevator::init(&address, elev_num_floors).expect("couldn't create elevator connection");

info!("Spawning threads.");
// spawn manager
let sender_tx_clone = sender_tx.clone();
let controller_tx_clone = controller_tx.clone();
let alarm_rx_clone = alarm_rx.clone();
let lights_tx_clone = lights_tx.clone();
let m = spawn(move || manager::run(
    id,
    manager_rx,
    sender_tx_clone,
    controller_tx_clone,
    lights_tx_clone,
    call_button_rx,
    alarm_rx_clone));
// spawn lights handler
let lights_rx_clone = lights_rx.clone();
let elev = elevator_connection.clone();
let l = spawn(move || lights::run(lights_rx_clone, elev));
// spawn controller
let manager_tx_clone = manager_tx.clone();
let elev = elevator_connection.clone();
let c = spawn(move || controller::run(controller_rx, manager_tx_clone, elev));
// spawn sender
let s = spawn(move || sender::run(sender_rx));
// spawn receiver
let manager_tx_clone = manager_tx.clone();
let r = spawn(move || receiver::run(manager_tx_clone));
// spawn call_buttons
let poll_period = Duration::from_millis(25);
let elev = elevator_connection.clone();
let b = spawn(move || elevio::poll::call_buttons(elev, call_button_tx, poll_period));
// spawn alarm
let timeout = Duration::from_secs(1);
let alarm_tx_clone = alarm_tx.clone();
let a = spawn(move || alarm::run(alarm_tx_clone, timeout));



// Test Block
// let mut init_requests = [[manager::RequestState::None;3]; config::FLOOR_COUNT];
// init_requests[0][2] = RequestState::Unconfirmed;
```

```
    // let wv = WorldView::init_with_requests(5, init_requests);
    // manager_tx.send(messages::Manager::HeartBeat(wv)).unwrap();

    // let mut init_requests = [[manager::RequestState::None;3]; config::FLOOR_COUNT];
    // init_requests[0][2] = RequestState::Confirmed;
    // let wv = WorldView::init_with_requests(5, init_requests);
    // manager_tx.send(messages::Manager::HeartBeat(wv)).unwrap();



    let _ = m.join();
    let _ = l.join();
    let _ = c.join();
    let _ = s.join();
    let _ = r.join();
    let _ = b.join();
    let _ = a.join();
}
```

Innhald frå Rust-filer

Fil: dbca849d/snapshot-[REDACTED]/main/src/controller.rs

```
use crossbeam_channel as cbc;
use driver_rust::elevio;
use driver_rust::elevio::elev as e;
use log::debug;
use log::info;

use crate::messages;
use crate::fsm;
use std::thread::spawn;
use std::time::Duration;

pub fn run(controller_rx: cbc::Receiver<messages::Controller>, manager_tx: cbc::Sender<messages::Manager>, elevator_connection: e::Elevator) -> std::io::Result<()> {
    info!("Controller up and running.");
    let (timer_tx, timer_rx) = cbc::unbounded::<bool>();
    let mut elevator_state = fsm::ElevatorState::init_elevator(elevator_connection.clone(), timer_tx);

    let poll_period = Duration::from_millis(25);

    info!("Starting hardware monitors.");
    let (floor_sensor_tx, floor_sensor_rx) = cbc::unbounded::<u8>();
    {
        let elevator = elevator_connection.clone();
        spawn(move || elevio::poll::floor_sensor(elevator, floor_sensor_tx, poll_period));
    }

    let (stop_button_tx, stop_button_rx) = cbc::unbounded::<bool>();
    {
        let elevator = elevator_connection.clone();
        spawn(move || elevio::poll::stop_button(elevator, stop_button_tx, poll_period));
    }

    let (obstruction_tx, obstruction_rx) = cbc::unbounded::<bool>();
    {
        let elevator = elevator_connection.clone();
        spawn(move || elevio::poll::obstruction(elevator, obstruction_tx, poll_period));
    }
    if elevator_connection.floor_sensor().is_none() {
        elevator_state.fsm_on_init_between_floors();
    }

    while elevator_connection.floor_sensor().is_none() {}

    loop {
        debug!("Waiting for input.");
        debug!("Before: {:?}", &elevator_state);
        cbc::select! {
            recv(controller_rx) -> a => {
                let message = a.unwrap();
                match message {
```

```
                messages::Controller::Requests(requests) => {
                    info!("Received Requests");
                    elevator_state.fsm_on_new_requests(requests, &manager_tx);
                }
            }
        },
        recv(floor_sensor_rx) -> a => {
            info!("Received FloorSensor");
            let floor_sensor = a.unwrap();
            elevator_state.fsm_on_floor_arrival(floor_sensor as i8, &manager_tx);
        },
        recv(stop_button_rx) -> a => {
            let _stop_button = a.unwrap();
            elevator_state.fsm_on_stop_button_press();
        },
        recv(obstruction_rx) -> a => {
            info!("Received Obstruction");
            let obstruction = a.unwrap();
            elevator_state.fsm_on_obstruction(obstruction);
        },
        recv(timer_rx) -> a => {
            info!("Received Timeout");
            let _time_out = a.unwrap();
            elevator_state.fsm_on_door_time_out(&manager_tx);
        }
    };
    debug!("After: {:?}", &elevator_state);
  }
}
```

Innhald frå Rust-filer

Fil: dbca849d/snapshot-[REDACTED]/main/src/config.rs

```
pub const FLOOR_COUNT: usize = 4;
pub const CALL_COUNT: usize = 3;
```

Innhald frå Rust-filer

Fil: dbca849d/snapshot-[REDACTED]/main/src/sender.rs

```
use core::net::SocketAddr;
use std::net::UdpSocket;

use crossbeam_channel as cbc;
use log::{debug, info};

use crate::messages;
use bincode;

pub fn run(rx: cbc::Receiver<messages::Manager>) {
    debug!("Sender up and running...");
    let addr: SocketAddr = "0.0.0.0:0".parse().unwrap();
    let destination_addr: SocketAddr = "0.0.0.0:4567".parse().unwrap();
    let socket = UdpSocket::bind(addr).unwrap();

    info!("Sending on {}", socket.local_addr().unwrap());

    loop {
        debug!("Waiting for input...");
        cbc::select! {
            recv(rx) -> a => {
                let packet = a.unwrap();
                let serialized = bincode::serialize(&packet).unwrap();
                socket.send_to(&serialized, destination_addr).unwrap();
            }
        }
    }
}
```

Innhald frå Rust-filer

Fil: dbca849d/snapshot-[REDACTED]/main/src/lights.rs

```
use crossbeam_channel as cbc;
use log::{trace, debug, info};
use crate::messages;
use crate::fsm;
use driver_rust::elevio::elev as e;
use crate::config;

pub fn run(lights_rx: cbc::Receiver<messages::Controller>, elev_conn: e::Elevator) {
    loop {
        cbc::select! {
            recv(lights_rx) -> a => {
                match a.unwrap() {
                    messages::Controller::Requests(requests) => {
                        info!("Received Requests");
                        debug!("{:?}", &requests);
                        set_all_lights(&elev_conn, &requests);
                    }
                }
            }
        }
    }
}
fn set_all_lights(elev_conn: &e::Elevator, requests: &fsm::ControllerRequests) {
    trace!("set_all_lights");
    for f in 0..config::FLOOR_COUNT {
        for b in 0..config::CALL_COUNT {
            elev_conn.call_button_light(f as u8, b as u8, requests[f as usize][b as usize]);
        }
    }
}
```

Innhald frå Rust-filer

Fil: dbca849d/snapshot-[REDACTED]/main/src/receiver.rs

```
use core::net::SocketAddr;
use std::net::UdpSocket;

use crossbeam_channel as cbc;
use log::{debug, info};

use crate::messages;

pub fn run(manager_tx: cbc::Sender<messages::Manager>) {
    debug!("Receiver up and running...");
    let addr: SocketAddr = "0.0.0.0:4567".parse().unwrap();

    let socket = UdpSocket::bind(addr).unwrap();
    info!("Listening on {}", socket.local_addr().unwrap());

    let mut buf = [0u8; 1024];

    loop {
        debug!("Ready for input...");
        let (_, _) = socket.recv_from(&mut buf).unwrap();
        // Deserialize the binary data back to a struct
        let deserialized: messages::Manager = bincode::deserialize(&buf).unwrap();
        manager_tx.send(deserialized).unwrap();
    }
}
```

Innhald frå Rust-filer

Fil: dbca849d/snapshot-[REDACTED]/main/src/alarm.rs

```
use core::time::Duration;
use std::thread;

use crossbeam_channel as cbc;
use log::debug;
pub fn run(alarm_tx: cbc::Sender<u8>, timeout: Duration) {
    loop {
        debug!("Going to sleep");
        thread::sleep(timeout);
        debug!("Sending alarm");
        alarm_tx.send(0).unwrap();
    }
}
```

Innhald frå Rust-filer

Fil: dbca849d/snapshot-[REDACTED]/main/src/messages.rs

```
use serde::{Serialize, Deserialize};
use crate::manager;
use crate::fsm;
#[derive(Debug, Serialize, Deserialize)]
pub enum Manager {
    Ping,
    HeartBeat(manager::WorldView),
    ElevatorState(fsm::Dirn, fsm::ElevatorBehaviour, i8),
    ClearRequest(usize, [bool; 3]) //floor
}

#[derive(Debug)]
pub enum Controller {
    Requests(fsm::ControllerRequests)
}
```

Innhald frå Rust-filer

Fil: dbca849d/snapshot-[REDACTED]/main/src/fsm.rs

```
use driver_rust::elevio::elev::Elevator;
use log::trace;
use serde::{Serialize, Deserialize};

use std::thread;
use std::time::Duration;
use crossbeam_channel::{self as cbc, Sender};
use crate::{config, messages};

const CALL_COUNT: usize = 3;
#[derive(Debug, Serialize, Deserialize, Clone, Copy)]
pub enum ElevatorBehaviour {
    Idle,
    DoorOpen,
    Moving
}
#[derive(Debug, Copy, Clone, Serialize, Deserialize)]
pub enum Dirn {
    Down = -1,
    Stop = 0,
    Up = 1
}
#[derive(Debug)]
enum Button {
    HallUp,
    HallDown,
    Cab
}
pub type ControllerRequests = [[bool;CALL_COUNT]; config::FLOOR_COUNT];
#[derive(Debug)]
pub struct ElevatorState {
    timer_tx: cbc::Sender<bool>,
    no_of_timer_threads: u8,
    floor: i8,
    dirn: Dirn,
    requests: ControllerRequests,
    behaviour: ElevatorBehaviour,
    door_open_duration: u64,
    connection: Elevator,
    obstruction: bool
}
#[derive(Debug)]
struct DirectionBehaviourPair {
    dirn: Dirn,
    behavior: ElevatorBehaviour
}

impl ElevatorState {

    pub fn init_elevator(elevator_connection: Elevator, timer_tx: cbc::Sender<bool>) -> ElevatorState {
```

```
        trace!("init_elevator");
        ElevatorState {
            timer_tx,
            no_of_timer_threads: 0,
            floor: -1,
            dirn: Dirn::Stop,
            requests: [[false;CALL_COUNT]; config::FLOOR_COUNT],
            behaviour: ElevatorBehaviour::Idle,
            door_open_duration: 3,
            connection: elevator_connection,
            obstruction: false
        }
    }
    pub fn fsm_on_new_requests(&mut self, requests: ControllerRequests, manager_tx: &Sender<messages::Manager>)
{
        self.requests = requests;
        match self.behaviour {
            ElevatorBehaviour::Idle => {
                let direction_behavior_pair = self.requests_choose_direction();
                self.dirn = direction_behavior_pair.dirn;
                self.behaviour = direction_behavior_pair.behavior;
                match self.behaviour {
                    ElevatorBehaviour::Idle => {},
                    ElevatorBehaviour::DoorOpen => {
                        self.connection.door_light(true);
                        self.start_time_out_thread();
                        self.requests_clear_at_current_floor(&manager_tx);
                    },
                    ElevatorBehaviour::Moving => {
                        self.connection.motor_direction(self.dirn as u8);
                    }
                };
            },
            _ => ()
        }
    }
    pub fn fsm_on_init_between_floors(&mut self) {
        trace!("fsm_on_init_between_floors");
        //motor direction
        self.connection.motor_direction(Dirn::Down as u8);
        self.dirn = Dirn::Down;
        self.behaviour = ElevatorBehaviour::Moving;
    }

    pub fn fsm_on_door_time_out(&mut self, manager_tx: &Sender<messages::Manager>) {
        trace!("fsm_on_door_time_out");
        self.no_of_timer_threads -= 1;
        if self.no_of_timer_threads > 0 {return;}
        if self.obstruction  {
            self.start_time_out_thread();
            return;
        }
```

```
    match self.behaviour {
        ElevatorBehaviour::DoorOpen => {
            let pair: DirectionBehaviourPair = self.requests_choose_direction();
            self.dirn = pair.dirn;
            self.behaviour = pair.behavior;

            match self.behaviour {
                ElevatorBehaviour::DoorOpen => {
                    self.start_time_out_thread();
                    self.requests_clear_at_current_floor(&manager_tx);
                },
                ElevatorBehaviour::Moving | ElevatorBehaviour::Idle => {
                    self.connection.door_light(false);
                    self.connection.motor_direction(self.dirn as u8);
                }
            }
        },
        _ => {}
    }
}
pub fn fsm_on_obstruction(&mut self, val: bool) {
    trace!("fsm_on_obstruction");
    self.obstruction = val;
}
pub fn fsm_on_floor_arrival(&mut self, floor: i8, manager_tx: &Sender<messages::Manager>) {
    trace!("fsm_on_floor_arrival");
    self.floor = floor;
    self.connection.floor_indicator(self.floor as u8);

    match self.behaviour {
        ElevatorBehaviour::Moving => {
            if self.requests_should_stop() {
                self.connection.motor_direction(Dirn::Stop as u8);
                self.connection.door_light(true);
                self.requests_clear_at_current_floor(&manager_tx);
                self.start_time_out_thread();
                self.set_all_lights();
                self.behaviour = ElevatorBehaviour::DoorOpen;
            }
        }
        _ => {},
    };

    manager_tx.send(messages::Manager::ElevatorState(self.dirn, self.behaviour, self.floor)).unwrap();
}

pub fn fsm_on_stop_button_press(&mut self){}

fn start_time_out_thread(&mut self) {
    trace!("sleep");
    self.no_of_timer_threads += 1;
    let timer_tx_clone = self.timer_tx.clone();
```

```
    let duration = self.door_open_duration;
    thread::spawn(move || {
        thread::sleep(Duration::from_secs(duration));
        timer_tx_clone.send(true).unwrap();
    });
}

fn set_all_lights(&self) {
    trace!("set_all_lights");
    for f in 0..config::FLOOR_COUNT {
        for b in 0..CALL_COUNT {
            self.connection.call_button_light(f as u8, b as u8, self.requests[f as usize][b as usize]);
        }
    }
}

fn requests_choose_direction(&mut self) -> DirectionBehaviourPair {
    trace!("requests_choose_direction");
    match self.dirn {
        Dirn::Up => {
            if self.requests_above() {
                DirectionBehaviourPair {dirn: Dirn::Up, behavior: ElevatorBehaviour::Moving}
            } else if self.requests_here() {
                DirectionBehaviourPair {dirn: Dirn::Down, behavior: ElevatorBehaviour::DoorOpen}
            } else if self.requests_below() {
                DirectionBehaviourPair {dirn: Dirn::Down, behavior: ElevatorBehaviour::Moving}
            } else {
                DirectionBehaviourPair {dirn: Dirn::Stop, behavior: ElevatorBehaviour::Idle}
            }
        },
        Dirn::Down => {
            if self.requests_below() {
                DirectionBehaviourPair {dirn: Dirn::Down, behavior: ElevatorBehaviour::Moving}
            } else if self.requests_here() {
                DirectionBehaviourPair {dirn: Dirn::Up, behavior: ElevatorBehaviour::DoorOpen}
            } else if self.requests_above() {
                DirectionBehaviourPair {dirn: Dirn::Up, behavior: ElevatorBehaviour::Moving}
            } else {
                DirectionBehaviourPair {dirn: Dirn::Stop, behavior: ElevatorBehaviour::Idle}
            }
        },
        Dirn::Stop => {
            if self.requests_here() {
                DirectionBehaviourPair {dirn: Dirn::Stop, behavior: ElevatorBehaviour::DoorOpen}
            } else if self.requests_above() {
                DirectionBehaviourPair {dirn: Dirn::Up, behavior: ElevatorBehaviour::Moving}
            } else if self.requests_below() {
                DirectionBehaviourPair {dirn: Dirn::Down, behavior: ElevatorBehaviour::Moving}
            } else {
                DirectionBehaviourPair {dirn: Dirn::Stop, behavior: ElevatorBehaviour::Idle}
            }
        }
```

```
    }
}

fn requests_clear_at_current_floor(&mut self, manager_tx: &Sender<messages::Manager>) {
    trace!("requests_clear_at_current_floor");
    let mut should_clear = [false; 3];
    self.requests[self.floor as usize][Button::Cab as usize] = false;
    should_clear[Button::Cab as usize] = true;
    match self.dirn {
        Dirn::Up => {
            if !self.requests_above() && (self.requests[self.floor as usize][Button::HallUp as usize] == false) {
                self.requests[self.floor as usize][Button::HallDown as usize] = false;
                should_clear[Button::HallDown as usize] = true;
            }
            self.requests[self.floor as usize][Button::HallUp as usize] = false;
            should_clear[Button::HallUp as usize] = true;
        },
        Dirn::Down => {
            if !self.requests_below() && (self.requests[self.floor as usize][Button::HallDown as usize] == false) {
                self.requests[self.floor as usize][Button::HallUp as usize] = false;
                should_clear[Button::HallUp as usize] = true;
            }
            self.requests[self.floor as usize][Button::HallDown as usize] = false;
            should_clear[Button::HallDown as usize] = true;
        },
        Dirn::Stop => {
            self.requests[self.floor as usize][Button::HallUp as usize] = false;
            self.requests[self.floor as usize][Button::HallDown as usize] = false;
            should_clear[Button::HallUp as usize] = true;
            should_clear[Button::HallDown as usize] = true;
        }
    }
    manager_tx.send(messages::Manager::ClearRequest(self.floor as usize, should_clear)).unwrap();
}

fn requests_here(&self) -> bool {
    trace!("requests_here");
    for b in 0..CALL_COUNT {
        if self.requests[self.floor as usize][b as usize] {
            return true;
        }
    }
    return false;
}

fn requests_below(&self) -> bool {
    trace!("requests_below");
    for f in 0..self.floor {
        for b in 0..CALL_COUNT {
            if self.requests[f as usize][b as usize] {
                return true;
            }
```

```
            }
        }
        return false;
    }

    fn requests_above(&self) -> bool {
        trace!("requests_above");
        for f in ((self.floor+1) as usize)..config::FLOOR_COUNT {
            for b in 0..CALL_COUNT {
                if self.requests[f as usize][b as usize] {
                    return true;
                }
            }
        }
        return false;
    }

    fn requests_should_stop(&self) -> bool {
        trace!("requests_should_stop");
        match self.dirn {
            Dirn::Down => {
                self.requests[self.floor as usize][Button::HallDown as usize] == true||
                    self.requests[self.floor as usize][Button::Cab as usize] == true||
                    !self.requests_below()
            },
            Dirn::Up => {
                self.requests[self.floor as usize][Button::HallUp as usize] == true||
                    self.requests[self.floor as usize][Button::Cab as usize] == true||
                    !self.requests_above()
            },
            Dirn::Stop => {true}
        }
    }

}
```

Innhald frå Rust-filer

Fil: dbca849d/snapshot-[REDACTED]/main/src/manager.rs

```
use driver_rust::elevio::poll::CallButton;
use serde::{Deserialize, Serialize};
use core::time::Duration;
use std::collections::HashMap;
use std::time::SystemTime;

use crate::config;
use crate::fsm;
use crate::fsm::ControllerRequests;
use crate::fsm::Dirn;
use crate::fsm::ElevatorBehaviour;
use crate::messages;
use crossbeam_channel as cbc;
use driver_rust::elevio;
use log::{debug, info};

#[derive(Debug, Serialize, Deserialize, Clone, Copy)]
pub enum RequestState {
    None = 0,
    Unconfirmed = 1,
    //Barrier
    Confirmed = 2,
}
#[derive(Debug, Serialize, Deserialize, Clone, Copy)]
pub struct ElevatorNetworkState {
    dirn: fsm::Dirn,
    behaviour: fsm::ElevatorBehaviour,
    current_floor: i8,
}
impl ElevatorNetworkState {
    // pub fn get_dirn(&self) -> fsm::Dirn {
    //     self.dirn
    // }
    // pub fn get_behaviour(&self) -> fsm::ElevatorBehaviour {
    //     self.behaviour
    // }
    // pub fn get_current_floor(&self) -> i8 {
    //     self.current_floor
    // }
}
pub type ManagerRequests = [[RequestState; 3]; config::FLOOR_COUNT];
pub fn manager_to_controller_requests(manager_reqs: &ManagerRequests) -> ControllerRequests {
    let mut controller_requests: ControllerRequests = [[false; config::CALL_COUNT]; config::FLOOR_COUNT];
    for floor in 0..config::FLOOR_COUNT {
        for call in 0..config::CALL_COUNT {
            controller_requests[floor][call] = match manager_reqs[floor][call] {
                RequestState::Confirmed => true,
                _ => false
            };
        }
```

```
    }
    controller_requests
}
#[derive(Debug, Serialize, Deserialize, Clone)]
pub struct Elevator {
    last_received: SystemTime,
    state: ElevatorNetworkState,
    requests: ManagerRequests
}
impl Elevator {
    pub fn set_last_received(&mut self, new_val: SystemTime) {
        self.last_received = new_val;
    }
}
#[derive(Debug, Serialize, Deserialize, Clone)]
pub struct WorldView {
    id: u8,
    pub elevators: HashMap<u8, Elevator>,
}

impl WorldView {
    // pub fn init_with_requests(id: u8, init_requests: ManagerRequests) -> WorldView {
    //     let mut elevators = HashMap::new();
    //     let our_elevator = Elevator {
    //         last_received: SystemTime::now(),
    //         state: ElevatorNetworkState {
    //             dirn: fsm::Dirn::Stop,
    //             behaviour: fsm::ElevatorBehaviour::Idle,
    //             current_floor: -1,
    //         },
    //         requests: init_requests
    //     };
    //     elevators.insert(id, our_elevator);
    //     WorldView {
    //         id,
    //         elevators
    //     }
    // }
    pub fn init(id: u8) -> WorldView {
        let mut elevators = HashMap::new();
        let requests = [[RequestState::None; 3]; config::FLOOR_COUNT];
        let our_elevator = Elevator {
            last_received: SystemTime::now(),
            state: ElevatorNetworkState {
                dirn: fsm::Dirn::Stop,
                behaviour: fsm::ElevatorBehaviour::Idle,
                current_floor: -1,
            },
            requests
        };
        elevators.insert(id, our_elevator);
        WorldView {
```

```
        id,
        elevators
    }
}

pub fn handle_foreign_world_view(
    &mut self,
    foreign_world_view: WorldView
) {
    let current_time = SystemTime::now();

    let foreign_id = foreign_world_view.get_id();
    let foreign_elevators= foreign_world_view.get_elevators();

    // update local elevator for id
    if let Some(e) = foreign_elevators.get(&foreign_id) { // get foreign elevator
        // update local version
        self.elevators.insert(foreign_id, e.clone());
        let local_elevator = self.elevators.get_mut(&foreign_id).unwrap();
        local_elevator.set_last_received(current_time);
    }


    // add elevators that we dont already know of
    for key in foreign_world_view.elevators.keys() {
        if !self.elevators.contains_key(key) {
            let u = foreign_world_view.elevators.get(&key).unwrap();
            self.elevators.insert(*key, u.clone());
        }
    }

    // for each id, floor, direction update the counter based on our value and received value
    for (id, their_elevator) in foreign_world_view.elevators.iter() {
        if *id == self.id {continue;}
        let our_elevator = self.elevators.get_mut(id).unwrap();
        for floor in 0..config::FLOOR_COUNT {
            for dir in 0..3 {
                our_elevator.requests[floor][dir] = match their_elevator.requests[floor][dir] {
                    RequestState::None => match our_elevator.requests[floor][dir] {
                        RequestState::None => RequestState::None,
                        RequestState::Unconfirmed => RequestState::Unconfirmed,
                        RequestState::Confirmed => RequestState::None,
                    },
                    RequestState::Unconfirmed => match our_elevator.requests[floor][dir] {
                        RequestState::None => RequestState::Unconfirmed,
                        RequestState::Unconfirmed => RequestState::Unconfirmed,
                        RequestState::Confirmed => RequestState::Confirmed,
                    },
                    RequestState::Confirmed => match our_elevator.requests[floor][dir] {
                        RequestState::None => RequestState::None,
                        RequestState::Unconfirmed => RequestState::Confirmed,
                        RequestState::Confirmed => RequestState::Confirmed,
```

```
                },
            };
        }
    }
}

    self.merge();
}

pub fn merge(&mut self) {
    let mut new_requests: ManagerRequests = [[RequestState::None; 3]; config::FLOOR_COUNT];
    for floor in 0..config::FLOOR_COUNT {
        for dir in 0..3 {

            // store request state for floor/direction in tmp_vector
            let mut tmp_vector: Vec<RequestState> = Vec::new();
            for (id, elevator) in self.elevators.iter() {
                // only include alive elevators (and ourselves)
                if elevator.last_received.elapsed().unwrap() > Duration::from_secs(1) && *id != self.id {
                    continue;
                }

                tmp_vector.push(elevator.requests[floor][dir]);
            }

            let mut count = [0;3]; // counts the occurences of a state for floor/dir
            for val in tmp_vector.iter() {
                match val {
                    RequestState::None => {
                        count[0] += 1;
                    },
                    RequestState::Unconfirmed => {
                        count[1] += 1;
                    },
                    RequestState::Confirmed => {
                        count[2] += 1;
                    }
                }
            }
            // all at barrier
            if count[0] == 0 && count[2] == 0 { // [0 n 0]
                new_requests[floor][dir] = RequestState::Confirmed;
            } else {
                match self.elevators.get(&self.id).unwrap().requests[floor][dir] {
                    RequestState::None => {
                        if count[1] > 0 {
                            new_requests[floor][dir] = RequestState::Unconfirmed;
                        } else {
                            new_requests[floor][dir] = RequestState::None;
                        }
                    },
                    RequestState::Unconfirmed => {
```

```
                    if count[2] > 0 {
                        new_requests[floor][dir] = RequestState::Confirmed;
                    } else {
                        new_requests[floor][dir] = RequestState::Unconfirmed;
                    }
                },
                RequestState::Confirmed => {
                    if count[0] > 0 {
                        new_requests[floor][dir] = RequestState::None;
                    } else {
                        new_requests[floor][dir] = RequestState::Confirmed;
                    }
                }
            }
        }
    }
}


    // replace old hall requests with new hall requests
    self.elevators.get_mut(&self.id).unwrap().requests = new_requests;
}

pub fn handle_button_press(&mut self, button_press: &CallButton) {
     let new_value = match self.elevators.get(&self.id).unwrap().requests[button_press.floor as usize][button_press.call as usize] {
        RequestState::None => RequestState::Unconfirmed,
        RequestState::Unconfirmed => RequestState::Unconfirmed,
        RequestState::Confirmed => RequestState::Confirmed
    };
        self.elevators.get_mut(&self.id).unwrap().requests[button_press.floor as usize][button_press.call as usize] = new_value;

    //notify relevant subsystems
}

pub fn handle_elevator_state(&mut self, dirn: Dirn, behaviour: ElevatorBehaviour, floor: i8) {
    let elev = self.elevators.get_mut(&self.id).unwrap();
    elev.state.dirn = dirn;
    elev.state.behaviour = behaviour;
    elev.state.current_floor = floor;
}

pub fn handle_clear_request(&mut self, floor: usize, should_clear: &[bool; 3]) {
    let elev = self.elevators.get_mut(&self.id).unwrap();
    debug!("Clearing {:?}", &should_clear);
    for i in 0..3 {
        if should_clear[i] {
            elev.requests[floor][i] = RequestState::None;
        }
    }
}
```

```
    // Getters
    pub fn get_id(&self) -> u8 {
        self.id
    }
    pub fn get_elevators(&self) -> HashMap<u8, Elevator> {
        self.elevators.clone()
    }
}

pub fn run(
    id: u8,
    manager_rx: cbc::Receiver<messages::Manager>,
    sender_tx: cbc::Sender<messages::Manager>,
    controller_tx: cbc::Sender<messages::Controller>,
    lights_tx: cbc::Sender<messages::Controller>,
    call_button_rx: cbc::Receiver<elevio::poll::CallButton>,
    alarm_rx: cbc::Receiver<u8>
) {
    info!("Manager up and running...");
    let mut world_view = WorldView::init(id);
    loop {
        debug!("Waiting for input...");
        debug!("Before: {:#?}", &world_view);
        cbc::select! {
            recv(manager_rx) -> a => {
                let message = a.unwrap();
                match message {
                    messages::Manager::Ping => {
                        info!("Received Ping");
                    },
                    messages::Manager::HeartBeat(foreign_world_view) => {

                        if foreign_world_view.id != world_view.get_id() {
                            info!("Received HeartBeat from {}", foreign_world_view.get_id());

                            world_view.handle_foreign_world_view(foreign_world_view);

                            inform_everybody(
                                &world_view,
                                &sender_tx,
                                &controller_tx,
                                &lights_tx);
                        }
                    },
                    messages::Manager::ElevatorState(dirn, behaviour, floor) => {
                        info!("Received ElevatorState");

                        world_view.handle_elevator_state(dirn, behaviour, floor);

                        inform_everybody(
                            &world_view,
                            &sender_tx,
```

```
                    &controller_tx,
                    &lights_tx);
            },
            messages::Manager::ClearRequest(floor, should_clear) => {
                info!("Received ClearRequest");

                world_view.handle_clear_request(floor, &should_clear);

                inform_everybody(
                    &world_view,
                    &sender_tx,
                    &controller_tx,
                    &lights_tx);
            }
        }
    },
    recv(call_button_rx) -> a => {
        let button_press = a.unwrap();
        info!("Received CallButton");
        debug!("{:?}", button_press);

        world_view.handle_button_press(&button_press);

        inform_everybody(
            &world_view,
            &sender_tx,
            &controller_tx,
            &lights_tx);

    },
    recv(alarm_rx) -> _a => {
        info!("Received Alarm");

        world_view.merge();

        inform_everybody(
            &world_view,
            &sender_tx,
            &controller_tx,
            &lights_tx);
    }
  }
  debug!("After: {:#?}", &world_view);
 }
}


fn inform_everybody(
    world_view: &WorldView,
    sender_tx: &cbc::Sender<messages::Manager>,
    controller_tx: &cbc::Sender<messages::Controller>,
    lights_tx: &cbc::Sender<messages::Controller>
```

```
) {
    let manager_reqs: ManagerRequests = world_view.get_elevators().get(&world_view.get_id()).unwrap().requests;

    let world_view_clone = world_view.clone();
    sender_tx.send(messages::Manager::HeartBeat(world_view_clone)).unwrap();

    let controller_reqs = manager_to_controller_requests(&manager_reqs);
    controller_tx.send(messages::Controller::Requests(controller_reqs)).unwrap();
    lights_tx.send(messages::Controller::Requests(controller_reqs)).unwrap();
}
```

Innhald frå Rust-filer

Fil: dbca849d/snapshot-[REDACTED]/network/src/main.rs

```
use std::thread::spawn;
use crossbeam_channel as cbc;
mod network;

fn main() {
    // Spawn the threads
    let (sender_tx, sender_rx) = cbc::unbounded::<network::NetworkMessage>();
    let (decider_tx, decider_rx) = cbc::unbounded::<network::NetworkMessage>();

    let sender = spawn(|| network::sender(sender_rx));
    let receiver = spawn(|| network::receiver(decider_tx));
    let decider = spawn(|| network::decider(decider_rx, sender_tx));

    sender.join().unwrap();
    receiver.join().unwrap();
    decider.join().unwrap();
}
```

Innhald frå Rust-filer

Fil: dbca849d/snapshot-[REDACTED]/network/src/network.rs

```
use crossbeam_channel as cbc;
use serde::{Serialize, Deserialize};
use bincode;

use core::time::Duration;
use std::{net::{SocketAddr, UdpSocket}, thread};

#[derive(Debug, Serialize, Deserialize)]
pub enum NetworkMessage {
    Decider(u8),
    Sender(u8),
    Receiver(u8)
}

pub fn sender(rx: cbc::Receiver<NetworkMessage>) {
    // Define the address and port to bind the socket to
    let addr: SocketAddr = "0.0.0.0:0".parse().unwrap();
    let destination_addr: SocketAddr = "0.0.0.0:4567".parse().unwrap();
    // Create the UDP socket
    let socket = UdpSocket::bind(addr).unwrap();
    println!("Sending on {}", socket.local_addr().unwrap());

    // Buffer to store incoming data


    // Loop to receive and process packets
    loop {
        println!("sender: Waiting for input...");
        cbc::select! {
            recv(rx) -> a => {
                let packet = a.unwrap();
                let serialized = bincode::serialize(&packet).unwrap();
                println!("sender: serialized: {:?}", serialized);
                socket.send_to(&serialized, destination_addr).unwrap();
            }
        }
    }

}
pub fn receiver(decider_tx: cbc::Sender<NetworkMessage>) {
    let addr: SocketAddr = "0.0.0.0:4567".parse().unwrap();

    let socket = UdpSocket::bind(addr).unwrap();
    println!("Listening on {}", socket.local_addr().unwrap());

    let mut buf = [0u8; 1024];

    loop {
        println!("receiver: Waiting for input...");
        let (_, _) = socket.recv_from(&mut buf).unwrap();
```

```
        // Deserialize the binary data back to a struct
        let deserialized: NetworkMessage = bincode::deserialize(&buf).unwrap();
        println!("receiver: deserialized {:?}", deserialized);
        decider_tx.send(deserialized).unwrap();
    }
}
pub fn decider(rx: cbc::Receiver<NetworkMessage>, sender_tx: cbc::Sender<NetworkMessage>) {
    let mut counter = 0;
    loop {
        thread::sleep(Duration::from_secs(1));
        println!("decider: sending message");
        sender_tx.send(NetworkMessage::Sender(counter)).unwrap();

        cbc::select! {
            recv(rx) -> a => {
                let packet = a.unwrap();
                println!("decider: local({}) packet({:?})", counter, packet)
            }
        }
        counter += 1;
    }
}
```