Innhald frå Rust-filer

Fil: elevator_pro\src\config.rs

```
//! Globale verdier osv
use std::net::Ipv4Addr;
use std::time::Duration;

pub static NETWORK_PREFIX: &str = "10.100.23"; //Hardkoda subnet må vel vere greit. DEt er jo ekstra sikkerheit

pub static PN_PORT: u16 = u16::MAX; // Port for TCP mellom mastere
pub static BCU_PORT: u16 = 50000; // Port for TCP mellom lokal master/backup
pub static DUMMY_PORT: u16 = 42069; // Port fro sending / mottak av UDP broadcast

pub static BC_LISTEN_ADDR: &str = "0.0.0.0";
pub static BC_ADDR: &str = "255.255.255.255";
pub static OFFLINE_IP: Ipv4Addr = Ipv4Addr::new(69, 69, 69, 69);

pub static LOCAL_ELEV_IP: &str = "localhost:15657";
pub const DEFAULT_NUM_FLOORS: u8 = 4;
pub const ELEV_POLL: Duration = Duration::from_millis(25);

pub const ERROR_ID: u8 = 255;

pub const MASTER_IDX: usize = 1;
pub const KEY_STR: &str = "Gruppe 25";

pub const TCP_TIMEOUT: u64 = 5000; // i millisekunder
pub const TCP_PER_U64: u64 = 10; // i millisekunder
pub const UDP_PERIOD: Duration = Duration::from_millis(TCP_PER_U64);
pub const TCP_PERIOD: Duration = Duration::from_millis(TCP_PER_U64);

pub const SLAVE_TIMEOUT: Duration = Duration::from_millis(100);

pub const UDP_BUFFER: usize = u16::MAX as usize;


/* Debug modes */
pub static mut PRINT_WV_ON: bool = true;
pub static mut PRINT_ERR_ON: bool = true;
pub static mut PRINT_WARN_ON: bool = true;
pub static mut PRINT_OK_ON: bool = true;
pub static mut PRINT_INFO_ON: bool = true;
pub static mut PRINT_ELSE_ON: bool = true;
```

Innhald frå Rust-filer

Fil: elevator_pro\src\init.rs

```
use core::time;
use std::sync::atomic::Ordering;

use crate::world_view::world_view::{self, serialize_worldview, ElevatorContainer, WorldView};
use crate::utils::{self, ip2id, print_err};
use crate::world_view::world_view::Task;
use env_logger::init;
use local_ip_address::local_ip;
use crate::world_view::world_view::TaskStatus;
use crate::config;

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

use std::env;

//Initialiserer worldview
pub async fn initialize_worldview() -> Vec<u8> {
    let mut worldview = WorldView::default();
    let mut elev_container = ElevatorContainer::default();
    let init_task = Task{
        id: 69,
        to_do: 0,
        status: TaskStatus::PENDING,
        is_inside: true,
    };
    elev_container.tasks.push(init_task.clone());
    elev_container.tasks_status.push(init_task.clone());

    // Hent lokal IP-adresse
    let ip = match local_ip() {
        Ok(ip) => ip,
        Err(e) => {
            print_err(format!("Fant ikke IP i starten av main: {}", e));
            panic!();
        }
    };

    // Hent ut egen ID (siste tall i IP-adressen)
    utils::SELF_ID.store(ip2id(ip), Ordering::SeqCst); // Seigast
    elev_container.elevator_id = utils::SELF_ID.load(Ordering::SeqCst);
    worldview.master_id = utils::SELF_ID.load(Ordering::SeqCst);
    worldview.add_elev(elev_container.clone());


    //Hør etter UDP i 1 sek?. Hvis den får en wordlview: oppdater
    let wv_from_udp = check_for_udp().await;
    if wv_from_udp.is_empty(){
        utils::print_info("Ingen andre på Nett".to_string());
        return serialize_worldview(&worldview);
    }

    //Hvis det er UDP-er på nettverker, koble deg til dem ved å sette worldview = dem sin + egen heis
    let mut wv_from_udp_deser = world_view::deserialize_worldview(&wv_from_udp);
    wv_from_udp_deser.add_elev(elev_container.clone());

    //Sett egen ID som master_ID hvis tidligere master har høyere ID enn deg
    if wv_from_udp_deser.master_id > utils::SELF_ID.load(Ordering::SeqCst) {
        wv_from_udp_deser.master_id = utils::SELF_ID.load(Ordering::SeqCst);
    }


    world_view::serialize_worldview(&wv_from_udp_deser)
}




/// Hører etter UDP broadcaster i 1 sekund
///
/// Passer på at UDPen er fra 'vårt' nettverk før den 'aksepterer' den for retur
pub async fn check_for_udp() -> Vec<u8> {
    let broadcast_listen_addr = format!("{}:{}", config::BC_LISTEN_ADDR, config::DUMMY_PORT);
    let socket_addr: SocketAddr = broadcast_listen_addr.parse().expect("Ugyldig adresse");
    let socket_temp = Socket::new(Domain::IPV4, Type::DGRAM, None).expect("Feil å lage ny socket  iinit");


    socket_temp.set_reuse_address(true).expect("feil i set_resuse_addr i init");
    socket_temp.set_broadcast(true).expect("Feil i set broadcast i init");
    socket_temp.bind(&socket_addr.into()).expect("Feil i bind i init");
    let socket = UdpSocket::from_std(socket_temp.into()).expect("Feil å lage socket i init");
    let mut buf = [0; config::UDP_BUFFER];
    let mut read_wv: Vec<u8> = Vec::new();

    let mut message: Cow<'_, str> = std::borrow::Cow::Borrowed("a");

    let time_start = Instant::now();
    let duration = Duration::from_secs(1);

    while Instant::now().duration_since(time_start) < duration {
        let recv_result = timeout(duration, socket.recv_from(&mut buf)).await;

        match recv_result {
            Ok(Ok((len, _))) => {
                message = String::from_utf8_lossy(&buf[..len]).into_owned().into();
            }
            Ok(Err(e)) => {
                utils::print_err(format!("udp_broadcast.rs, udp_listener(): {}", e));
                continue;
            }
            Err(_) => {
                // Timeout skjedde  stopp løkka
                utils::print_warn("Timeout  ingen data mottatt innen 1 sekund.".to_string());
                break;
            }
        }

        // verifiser at UDPen er fra 'oss'
        if &message[1..config::KEY_STR.len() + 1] == config::KEY_STR {
            let clean_message = &message[config::KEY_STR.len() + 3..message.len() - 1]; // Fjerner `"`
            read_wv = clean_message
                .split(", ") // Del opp på ", "
                .filter_map(|s| s.parse::<u8>().ok()) // Konverter til u8, ignorer feil
                .collect(); // Samle i Vec<u8>

            break;
        }
    }
    drop(socket);
    read_wv
}


/// ### Leser argumenter på cargo run
///
/// Brukes for å endre hva som printes i runtime. Valgmuligheter:
///
/// `print_wv::(true/false)` &rarr; Printer worldview 2 ganger i sekundet
/// `print_err::(ture/false)` &rarr; Printer error meldinger
/// `print_wrn::(true/false)` &rarr; Printer warning meldinger
/// `print_ok::(true/false)` &rarr; Printer ok meldinger
/// `print_info::(true/false)` &rarr; Printer info meldinger
/// `print_else::(true/false` &rarr; Printer andre meldinger, bla. master, slave, color meldinger
/// `debug::` &rarr; Skrur av alle prints andre enn error meldinger
/// `help` &rarr; Skriver ut alle mulige argumenter uten å starte programmet
///
/// Alle prints er på om ingen argumnter er gitt
pub fn parse_args() {
    let args: Vec<String> = env::args().collect();

    if args.len() > 1 {
        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";

                unsafe {
                    match key.as_str() {
                        "print_wv" => config::PRINT_WV_ON = is_true,
                        "print_err" => config::PRINT_ERR_ON = is_true,
                        "print_warn" => config::PRINT_WARN_ON = is_true,
                        "print_ok" => config::PRINT_OK_ON = is_true,
                        "print_info" => config::PRINT_INFO_ON = is_true,
                        "print_else" => config::PRINT_ELSE_ON = is_true,
                        "debug" => { // Debug modus: Kun error-meldingar
                            config::PRINT_WV_ON = false;
                            config::PRINT_WARN_ON = false;
                            config::PRINT_OK_ON = false;
                            config::PRINT_INFO_ON = false;
                            config::PRINT_ELSE_ON = 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)");
                std::process::exit(0);
            }
        }
    }
}
```

Innhald frå Rust-filer

Fil: elevator_pro\src\lib.rs

```
pub mod config;
pub mod utils;
pub mod init;

pub mod network{
    pub mod udp_broadcast;
    pub mod local_network;
    pub mod tcp_network;
    pub mod tcp_self_elevator;
}

pub mod world_view{
    pub mod world_view_ch;
    pub mod world_view_update;
    pub mod world_view;
}

pub mod elevio {
    pub mod elev;
    pub mod poll;
}

pub mod elevator_logic {
    pub mod task_handler;
    pub mod master {
        pub mod wv_from_slaves;
        pub mod task_allocater;
    }
}
```

Innhald frå Rust-filer

Fil: elevator_pro\src\main.rs

```
use std::{fmt::format, sync::atomic::Ordering, time::Duration};

use elevator_pro::{config, elevator_logic::master::task_allocater, network::{local_network, tcp_network,
tcp_self_elevator, udp_broadcast}, utils::{self, print_err, print_info, print_ok}, world_view::{world_view, world_view_ch,
world_view_update}};
use elevator_pro::init;

use tokio::{sync::broadcast, time::sleep};
use tokio::sync::mpsc;
use tokio::net::TcpStream;
use std::net::SocketAddr;



#[tokio::main]
async fn main() {
    // Oppdater config-verdier basert på argumenter
    init::parse_args();

/* START ----------- Task for å overvake Nettverksstatus ---------------------- */
    /* oppdaterer ein atomicbool der true er online, false er då offline */
    let _network_status_watcher_task = tokio::spawn(async move {
        utils::print_info("Starter å passe på nettverket".to_string());
        let _ = world_view_update::watch_ethernet().await;
    });
/* SLUTT ----------- Task for å overvake Nettverksstatus ---------------------- */



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

/* START ----------- Init av lokale channels ---------------------- */
    //Kun bruk mpsc-rxene fra main_local_chs
    let main_local_chs = local_network::LocalChannels::new();
    let _ = main_local_chs.watches.txs.wv.send(worldview_serialised.clone());
/* SLUTT ----------- Init av lokale channels ---------------------- */



/* START ----------- Kloning av lokale channels til Tokio Tasks ---------------------- */
    let chs_udp_listen = main_local_chs.clone();
    let chs_udp_bc = main_local_chs.clone();
    let chs_tcp = main_local_chs.clone();
    let chs_udp_wd = main_local_chs.clone();
    let chs_print = main_local_chs.clone();
    let chs_listener = main_local_chs.clone();
    let chs_local_elev = main_local_chs.clone();
    let chs_task_allocater = main_local_chs.clone();
    let mut chs_loop = main_local_chs.clone();
    let (socket_tx, socket_rx) = mpsc::channel::<(TcpStream, SocketAddr)>(100);
/* SLUTT ----------- Kloning av lokale channels til Tokio Tasks ---------------------- */

/* START ----------- Starte kritiske tasks ----------- */
    //Task som kontinuerlig oppdaterer lokale worldview
    let _update_wv_task = tokio::spawn(async move {
        utils::print_info("Starter å oppdatere wv".to_string());
        let _ = world_view_ch::update_wv(main_local_chs, worldview_serialised).await;
    });
    //Task som håndterer den lokale heisen
    //TODO: Få den til å signalisere at vi er i known state.
    let _local_elev_task = tokio::spawn(async {
        let _ = tcp_self_elevator::run_local_elevator(chs_local_elev).await;
    });
/* SLUTT ----------- Starte kritiske tasks ----------- */



/* START ----------- Starte Eksterne Nettverkstasks ---------------------- */
    //Task som hører etter UDP-broadcasts
    let _listen_task = tokio::spawn(async move {
        utils::print_info("Starter å høre etter UDP-broadcast".to_string());
        let _ = udp_broadcast::start_udp_listener(chs_udp_listen).await;
    });
    //Task som starter egen UDP-broadcaster
    let _broadcast_task = tokio::spawn(async move {
        utils::print_info("Starter UDP-broadcaster".to_string());
        let _ = udp_broadcast::start_udp_broadcaster(chs_udp_bc).await;
    });
    //Task som håndterer TCP-koblinger
    let _tcp_task = tokio::spawn(async move {
        utils::print_info("Starter å TCPe".to_string());
        let _ = tcp_network::tcp_handler(chs_tcp, socket_rx).await;
    });
    //UDP Watchdog
    let _udp_watchdog = tokio::spawn(async move {
        utils::print_info("Starter udp watchdog".to_string());
        let _ = udp_broadcast::udp_watchdog(chs_udp_wd).await;
    });
    //Task som starter TCP-listener
    let _listener_handle = tokio::spawn(async move {
        utils::print_info("Starter tcp listener".to_string());
        let _ = tcp_network::listener_task(chs_listener, socket_tx).await;
    });
    //Task som fordeler heis-tasks
    let _allocater_handle = tokio::spawn(async move {
        utils::print_info("Starter task allocater listener".to_string());
        let _ = task_allocater::distribute_task(chs_task_allocater).await;
    });
    // Lag prat med egen heis thread her
/* SLUTT ----------- Starte Eksterne Nettverkstasks ---------------------- */

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

    //Vent med å avslutte programmet
    let _ = chs_loop.broadcasts.rxs.shutdown.recv().await;
}
```

Innhald frå Rust-filer

Fil: elevator_pro\src\utils.rs

```
use std::time::Duration;
use std::{fmt::format, io::Write};
use std::net::IpAddr;
use std::u8;
use tokio::net::TcpStream;
use tokio::io::AsyncWriteExt;
use termcolor::{Color, ColorChoice, ColorSpec, StandardStream, WriteColor};
use tokio::time::sleep;
use crate::{config, network::local_network, world_view::world_view::{self, Task}};

use local_ip_address::local_ip;

use std::sync::atomic::{AtomicU8, Ordering};

// Definer ein global `AtomicU8`
pub static SELF_ID: AtomicU8 = AtomicU8::new(u8::MAX); // Startverdi 0




/// Returnerer kommando for å åpne terminal til tilhørende OS
///
/// # Eksempel
/// ```
/// let (cmd, args) = get_terminal_command();
/// ```
/// returnerer:
///
/// linux -> "gnome-terminal", "--""
///
/// windows ->  "cmd", "/C", "start"
pub fn get_terminal_command() -> (String, Vec<String>) {
    // Detect platform and return appropriate terminal command
    if cfg!(target_os = "windows") {
        ("cmd".to_string(), vec!["/C".to_string(), "start".to_string()])
    } else {
        ("gnome-terminal".to_string(), vec!["--".to_string()])
    }
}

/// Returnerer lokal IPv4-addresse til maskinen som `IpAddr`
///
/// Om lokal IPv4-addresse ikke fins, returneres `local_ip_address::Error`
pub fn get_self_ip() -> Result<IpAddr, local_ip_address::Error> {
    let ip = match local_ip() {
        Ok(ip) => {
            ip
        }
        Err(e) => {
            print_warn(format!("Fant ikke IP i get_self_ip() -> Vi er offline: {}", e));
```

```
            return Err(e);
        }
    };
    Ok(ip)
}


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

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

/// Henter roten av IPen
///
/// # Eksempel
/// ```
/// let id = id_fra_ip("a.b.c.d");
/// ```
/// returnerer "a.b.c"
///
pub fn get_root_ip(ip: IpAddr) -> String {
    match ip {
        IpAddr::V4(addr) => {
            let octets = addr.octets();
            format!("{}.{}.{}", octets[0], octets[1], octets[2])
        }
        IpAddr::V6(addr) => {
            let segments = addr.segments();
            let root_segments = &segments[..segments.len() - 1]; // Fjern siste segment
```

```
            root_segments.iter().map(|s| s.to_string()).collect::<Vec<_>>().join(":")
        }
    }
}



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

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

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

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

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

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

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

Innhald frå Rust-filer

```
/// ### Printes når noe skjer som i teorien er logisk umulig
pub fn print_cosmic_err(fun: String) {
    let mut stdout = StandardStream::stdout(ColorChoice::Always);
    // Skriv ut "[ERROR]:" i rød
    stdout.set_color(ColorSpec::new().set_fg(Some(Color::Red))).unwrap();
    write!(&mut stdout, "[ERROR]: ").unwrap();
    // Definer regnbuefargene
    let colors = [
        Color::Red,
        Color::Yellow,
        Color::Green,
        Color::Cyan,
        Color::Blue,
        Color::Magenta,
    ];
    // Resten av meldingen i regnbuefarger
    let message = format!("Cosmic rays flipped a bit!    1 0 IN: {}", fun);
    for (i, c) in message.chars().enumerate() {
        let color = colors[i % colors.len()];
        stdout.set_color(ColorSpec::new().set_fg(Some(color))).unwrap();
        write!(&mut stdout, "{}", c).unwrap();
    }
    // Tilbakestill fargen
    stdout.set_color(&ColorSpec::new()).unwrap();
    println!();
}

/// Henter klone av nyeste wv i systemet
pub fn get_wv(chs: local_network::LocalChannels) -> Vec<u8> {
    chs.watches.rxs.wv.borrow().clone()
}

/// Oppdaterer `wv` til den nyeste lokale worldviewen
///
/// Oppdaterer kun om worldview er endra siden forrige gang funksjonen ble kalla
pub async fn update_wv(mut chs: local_network::LocalChannels, wv: &mut Vec<u8>) -> bool {
    if chs.watches.rxs.wv.changed().await.is_ok() {
        *wv = chs.watches.rxs.wv.borrow().clone();
        return true
    }
    false
}

/// Sjekker om du er master, basert på nyeste worldview
pub fn is_master(/*chs: local_network::LocalChannels */wv: Vec<u8>) -> bool {
    // let wv: Vec<u8> = get_wv(chs.clone());
    return SELF_ID.load(Ordering::SeqCst) == wv[config::MASTER_IDX];
}

pub fn get_elev_tasks(chs: local_network::LocalChannels) -> Vec<Task> {
    chs.watches.rxs.elev_task.borrow().clone()
}
```

Innhald frå Rust-filer

```
/// Henter klone av elevator_container med `id` fra nyeste worldview
pub fn extract_elevator_container(wv: Vec<u8>, id: u8) -> world_view::ElevatorContainer {
    let mut deser_wv = world_view::deserialize_worldview(&wv);

    deser_wv.elevator_containers.retain(|elevator| elevator.elevator_id == id);
    deser_wv.elevator_containers[0].clone()
}

/// Henter klone av elevator_container med `SELF_ID` fra nyeste worldview
pub fn extract_self_elevator_container(wv: Vec<u8>) -> world_view::ElevatorContainer {
    extract_elevator_container(wv, SELF_ID.load(Ordering::SeqCst))
}



pub async fn close_tcp_stream(stream: &mut TcpStream) {
    // Hent IP-adresser
    let local_addr = stream.local_addr().map_or_else(
        |e| format!("Ukjent (Feil: {})", e),
        |addr| addr.to_string(),
    );

    let peer_addr = stream.peer_addr().map_or_else(
        |e| format!("Ukjent (Feil: {})", e),
        |addr| addr.to_string(),
    );

    // Prøv å stenge streamen (Asynkront)
    match stream.shutdown().await {
        Ok(_) => print_info(format!(
            "TCP-forbindelsen er avslutta korrekt: {} -> {}",
            local_addr, peer_addr
        )),
        Err(e) => print_err(format!(
            "Feil ved avslutting av TCP-forbindelsen ({} -> {}): {}",
            local_addr, peer_addr, e
        )),
    }
}


pub async fn slave_sleep() {
    let _ = sleep(config::SLAVE_TIMEOUT);
}
```

Innhald frå Rust-filer

Fil: elevator_pro\src\elevator_logic\task_handler.rs

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

use crate::network::local_network;
use crate::utils::update_wv;
use crate::world_view::world_view::{ElevatorContainer, TaskStatus};
use crate::elevio::elev;
use crate::{utils, world_view::world_view};


pub async fn execute_tasks(chs: local_network::LocalChannels, elevator: elev::Elevator){
    let mut wv = utils::get_wv(chs.clone());

    // loop{
    //     let wv = utils::get_wv(chs.clone());
    //     let wv_deser = world_view::deserialize_worldview(&wv);
    //     world_view::print_wv(wv);

    // }
    let mut container: ElevatorContainer;
    update_wv(chs.clone(), &mut wv).await;
    container = utils::extract_self_elevator_container(wv.clone());update_wv(chs.clone(), &mut wv).await;
    container = utils::extract_self_elevator_container(wv.clone());
    elevator.motor_direction(elev::DIRN_DOWN);

    loop {
        // let tasks_from_udp = utils::get_elev_tasks(chs.clone());
        update_wv(chs.clone(), &mut wv).await;
        container = utils::extract_self_elevator_container(wv.clone());
        let tasks_from_udp = container.tasks;
        // utils::print_err(format!("last_floor: {}", container.last_floor_sensor));
        if !tasks_from_udp.is_empty() {
            //utils::print_err(format!("TODO: {}, last_floor: {}", 0, container.last_floor_sensor));
            if tasks_from_udp[0].to_do < container.last_floor_sensor {
                elevator.motor_direction(elev::DIRN_DOWN);
            }
            else if tasks_from_udp[0].to_do > container.last_floor_sensor {
                elevator.motor_direction(elev::DIRN_UP);
            }
            else {
                elevator.motor_direction(elev::DIRN_STOP);
                // Si fra at første task er ferdig
                let _ = chs.mpscs.txs.update_task_status.send((tasks_from_udp[0].id, TaskStatus::DONE)).await;
                // open_door_protocol().await;
                sleep(Duration::from_millis(3000));
            }
        }
    }
}
```

Innhald frå Rust-filer

Fil: elevator_pro\src\elevator_logic\master\task_allocater.rs

```
//! # Denne delen av prosjektet er 'ikke påbegynt'
use std::{thread::sleep, time::Duration};

use crate::{elevio::poll::{CallButton, CallType}, network::local_network, utils, world_view::world_view::{self, ElevatorContainer, Task, TaskStatus}};


struct Orders {
    task: Vec<Task>,
}

/// ### Ikke ferdig, såvidt starta
///
/// Nå gir den task som er feil til feil heis !
pub async fn distribute_task(chs: local_network::LocalChannels) {
    let mut i: u16 = 0;
    let mut wv = utils::get_wv(chs.clone());
    let mut wv_deser = world_view::deserialize_worldview(&wv);
    let mut prev_button_0 = CallButton{call: CallType::from(69), floor: 255, elev_id: 255};

    loop {
        utils::update_wv(chs.clone(), &mut wv).await;

        while utils::is_master(wv.clone()) {
            utils::update_wv(chs.clone(), &mut wv).await;
            wv_deser = world_view::deserialize_worldview(&wv);
            let buttons = wv_deser.outside_button;

            if !buttons.is_empty() && buttons[0] != prev_button_0 {
                let task = create_task(buttons[0], i);
                i = (i % (u16::MAX - 1000)) + 1;
                let (mut lowest_cost, mut id) = (i32::MAX, 0);

                for elev in wv_deser.elevator_containers.iter() {
                    let cost = calculate_cost(task.clone(), elev.clone());
                    if cost < lowest_cost {
                        lowest_cost = cost;
                        id = elev.elevator_id;
                    }
                }
                let _ = chs.mpscs.txs.new_task.send((task, id, buttons[0])).await;
                println!("Antall knapper: {}", buttons.len());
                prev_button_0 = buttons[0];
            }
        }

        sleep(Duration::from_millis(100));
    }
}
```

Innhald frå Rust-filer

```
fn create_task(button: CallButton, task_id: u16) -> Task {
    Task { id: task_id, to_do: button.floor, status: TaskStatus::PENDING, is_inside: false }
}

fn calculate_cost(task: Task, elev: ElevatorContainer) -> i32 {
    elev.tasks.len() as i32
}


// fn optimze_active_tasks()



// ---------------------------------------------------------------------------
// Kalkulerer ein "kostnad" for kor godt ein heis kan ta imot eit eksternt kall
// ---------------------------------------------------------------------------
/*
fn kalkuler_kostnad(elev: &ElevatorStatus, call: &CallButton) -> u32 {
    // Basiskostnad er avstanden i etasjar
    let diff = if elev.current_floor > call.floor {
        elev.current_floor - call.floor
    } else {
        call.floor - elev.current_floor
    } as u32;
    let mut kostnad = diff;

    // Legg til ekstra kostnad dersom heisens retning ikkje stemmer med kallretninga
    match (elev.direction, call.call) {
        // Om heisen køyrer opp og kall er UP, og heisen er under kall-etasjen
        (Direction::Up, CallType::UP) if elev.current_floor <= call.floor => { }
        // Om heisen køyrer ned og kall er DOWN, og heisen er over kall-etasjen
        (Direction::Down, CallType::DOWN) if elev.current_floor >= call.floor => { }
        // Om heisen er idle er det optimalt
        (Direction::Idle, _) => { }
        // I alle andre tilfelle legg til ein straff
        _ => {
            kostnad += 100;
        }
    }

    // Legg til kostnad basert på talet på allereie tildelte oppgåver
    kostnad += (elev.tasks.len() as u32) * 10;

    kostnad
}

// ---------------------------------------------------------------------------
// Funksjon som tildeler ein oppgåve til rett heis
//
// - For INSIDE kall: finn heisen med samsvarande elev_id (forutsatt at han ikkje er offline).
// - For eksterne kall (UP/DOWN): vel heisen med lågaste kostnad.
// ---------------------------------------------------------------------------
pub fn tildele_oppgave(elevators: &[ElevatorStatus], call: CallButton) -> Option<u8> {
    // Dersom kalltypen er INSIDE, skal oppgåva gå til den spesifikke heisen
    if call.call == CallType::INSIDE {
        return elevators.iter()
        .find(|e| e.elevator_id == call.elev_id && !e.offline)
        .map(|e| e.elevator_id);
}

// For eksterne kall: iterer gjennom alle heisar som ikkje er offline
let mut beste_id = None;
let mut beste_kostnad = u32::MAX;

for elev in elevators.iter().filter(|e| !e.offline) {
    let kost = kalkuler_kostnad(elev, &call);
    if kost < beste_kostnad {
        beste_kostnad = kost;
        beste_id = Some(elev.elevator_id);
    }
}

beste_id
}

*/
```

Innhald frå Rust-filer

Fil: elevator_pro\src\elevator_logic\master\wv_from_slaves.rs

```
use crate::world_view::world_view::{self, ElevatorContainer};
use crate::elevator_logic::master::wv_from_slaves::world_view::TaskStatus;
use std::collections::HashSet;

/// ### Oppdatere statuser til slave-heis basert på melding fra TCP
pub async fn update_statuses(deser_wv: &mut world_view::WorldView, container: &ElevatorContainer, i: usize) {
    //Setter alle 'enkle' statuser likt som slaven har
    deser_wv.elevator_containers[i].door_open = container.door_open;
    deser_wv.elevator_containers[i].last_floor_sensor = container.last_floor_sensor;
    deser_wv.elevator_containers[i].obstruction = container.obstruction;
    deser_wv.elevator_containers[i].motor_dir = container.motor_dir;
    deser_wv.elevator_containers[i].calls = container.calls.clone();
    deser_wv.elevator_containers[i].tasks_status = container.tasks_status.clone();

    // Finner ID til tasks slaven er ferdig med
    let completed_tasks_ids: HashSet<u16> = container
        .tasks_status
        .iter()
        .filter(|t| t.status == TaskStatus::DONE)
        .map(|t| t.id)
        .collect();

    /*_____ Fjern Tasks som er markert som ferdig av slaven _____ */
    deser_wv.elevator_containers[i].tasks.retain(|t| !completed_tasks_ids.contains(&t.id));
}

/// ### Oppdaterer globale call_buttons fra slaven sine lokale call_buttons
pub async fn update_call_buttons(deser_wv: &mut world_view::WorldView, container: &ElevatorContainer, i: usize) {
    // Sett opp et HashSet for å sjekke for duplikater
    let mut seen = HashSet::new();

    // Legg til eksisterende elementer i HashSet
    for &elem in &deser_wv.outside_button.clone() {
        seen.insert(elem);
    }

    // Utvid outside_button med elementer som ikke er i HashSet
    //println!("Callbtwns hos slave {}: {:?}", container.elevator_id, container.calls);
    for &call in &container.calls {
        if !seen.contains(&call) {
            deser_wv.outside_button.push(call);
            seen.insert(call.clone());
        }
    }
}

/// Kommende funksjon
pub async fn update_tasks() {

}
```

Innhald frå Rust-filer

Fil: elevator_pro\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\src\elevio\poll.rs

```
use crossbeam_channel as cbc;
use std::sync::atomic::Ordering;
use std::thread;
use std::time;
use serde::{Serialize, Deserialize};
use std::hash::{Hash, Hasher};

use crate::utils;

use super::elev::{self, DIRN_STOP, DIRN_DOWN, DIRN_UP};

//Lager enum for call
#[derive(Serialize, Deserialize, Debug, Clone, Copy, PartialEq, Eq, Hash)]
pub enum CallType {
    UP = 0,
    DOWN,
    INSIDE,
    COSMIC_ERROR,
}
impl From<u8> for CallType {
    fn from(value: u8) -> Self {
        match value {
            0 => CallType::UP,
            1 => CallType::DOWN,
            2 => CallType::INSIDE,
            _ => {
                utils::print_cosmic_err("Call type does not exist".to_string());
                CallType::COSMIC_ERROR
            },
        }
    }
}
#[derive(Serialize, Deserialize, Debug, Clone, Copy, Eq)] // Added support for (De)serialization and cloning
pub struct CallButton {
    pub floor: u8,
    pub call: CallType,
    pub elev_id: u8,
}

impl PartialEq for CallButton {
    fn eq(&self, other: &Self) -> bool {
        // Hvis call er INSIDE, sammenligner vi også elev_id
        if self.call == CallType::INSIDE {
            self.floor == other.floor && self.call == other.call && self.elev_id == other.elev_id
        } else {
            // For andre CallType er det tilstrekkelig å sammenligne floor og call
            self.floor == other.floor && self.call == other.call
        }
    }
}
```

Innhald frå Rust-filer

```
impl Hash for CallButton {
    fn hash<H: Hasher>(&self, state: &mut H) {
        // Sørger for at hash er konsistent med eq
        self.floor.hash(state);
        self.call.hash(state);
        if self.call == CallType::INSIDE {
            self.elev_id.hash(state);
        }
    }
}

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 {
                    ch.send(CallButton { floor: f, call: CallType::from(c), elev_id:
utils::SELF_ID.load(Ordering::SeqCst)}).unwrap();
                }
                prev[f as usize][c as usize] = v;
            }
        }
        thread::sleep(period)
    }
}

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)
    }
}

pub fn stop_button(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

```
pub fn obstruction(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)
    }
}
```

Innhald frå Rust-filer

Fil: elevator_pro\src\network\local_network.rs

```
use crate::{elevio::poll::CallButton, world_view::world_view::TaskStatus};
use tokio::sync::{mpsc, broadcast, watch, Semaphore};
use std::sync::Arc;
use crate::world_view::world_view::Task;


#[derive(Debug)]
pub enum ElevMsgType {
    CBTN,
    FSENS,
    SBTN,
    OBSTRX,
}

#[derive(Debug)]
pub struct ElevMessage {
    pub msg_type: ElevMsgType,
    pub call_button: Option<CallButton>,
    pub floor_sensor: Option<u8>,
    pub stop_button: Option<bool>,
    pub obstruction: Option<bool>,
}


// --- MPSC-KANALAR ---

pub struct MpscTxs {
    pub udp_wv: mpsc::Sender<Vec<u8>>,
    pub tcp_to_master_failed: mpsc::Sender<bool>,
    pub container: mpsc::Sender<Vec<u8>>,
    pub remove_container: mpsc::Sender<u8>,
    pub local_elev: mpsc::Sender<ElevMessage>,
    pub sent_tcp_container: mpsc::Sender<Vec<u8>>,

    // 10 nye buffer-kanalar
    pub new_task: mpsc::Sender<(Task, u8, CallButton)>,
    pub update_task_status: mpsc::Sender<(u16, TaskStatus)>,
    pub mpsc_buffer_ch2: mpsc::Sender<Vec<u8>>,
    pub mpsc_buffer_ch3: mpsc::Sender<Vec<u8>>,
    pub mpsc_buffer_ch4: mpsc::Sender<Vec<u8>>,
    pub mpsc_buffer_ch5: mpsc::Sender<Vec<u8>>,
    pub mpsc_buffer_ch6: mpsc::Sender<Vec<u8>>,
    pub mpsc_buffer_ch7: mpsc::Sender<Vec<u8>>,
    pub mpsc_buffer_ch8: mpsc::Sender<Vec<u8>>,
    pub mpsc_buffer_ch9: mpsc::Sender<Vec<u8>>,
}

pub struct MpscRxs {
    pub udp_wv: mpsc::Receiver<Vec<u8>>,
    pub tcp_to_master_failed: mpsc::Receiver<bool>,
    pub container: mpsc::Receiver<Vec<u8>>,
    pub remove_container: mpsc::Receiver<u8>,
    pub local_elev: mpsc::Receiver<ElevMessage>,
    pub sent_tcp_container: mpsc::Receiver<Vec<u8>>,

    // 10 nye buffer-kanalar
    pub new_task: mpsc::Receiver<(Task, u8, CallButton)>,
    pub update_task_status: mpsc::Receiver<(u16, TaskStatus)>,
    pub mpsc_buffer_ch2: mpsc::Receiver<Vec<u8>>,
    pub mpsc_buffer_ch3: mpsc::Receiver<Vec<u8>>,
    pub mpsc_buffer_ch4: mpsc::Receiver<Vec<u8>>,
    pub mpsc_buffer_ch5: mpsc::Receiver<Vec<u8>>,
    pub mpsc_buffer_ch6: mpsc::Receiver<Vec<u8>>,
    pub mpsc_buffer_ch7: mpsc::Receiver<Vec<u8>>,
    pub mpsc_buffer_ch8: mpsc::Receiver<Vec<u8>>,
    pub mpsc_buffer_ch9: mpsc::Receiver<Vec<u8>>,
}

impl Clone for MpscTxs {
    fn clone(&self) -> MpscTxs {
        MpscTxs {
            udp_wv: self.udp_wv.clone(),
            tcp_to_master_failed: self.tcp_to_master_failed.clone(),
            container: self.container.clone(),
            remove_container: self.remove_container.clone(),
            local_elev: self.local_elev.clone(),
            sent_tcp_container: self.sent_tcp_container.clone(),

            // Klonar buffer-kanalane
            new_task: self.new_task.clone(),
            update_task_status: self.update_task_status.clone(),
            mpsc_buffer_ch2: self.mpsc_buffer_ch2.clone(),
            mpsc_buffer_ch3: self.mpsc_buffer_ch3.clone(),
            mpsc_buffer_ch4: self.mpsc_buffer_ch4.clone(),
            mpsc_buffer_ch5: self.mpsc_buffer_ch5.clone(),
            mpsc_buffer_ch6: self.mpsc_buffer_ch6.clone(),
            mpsc_buffer_ch7: self.mpsc_buffer_ch7.clone(),
            mpsc_buffer_ch8: self.mpsc_buffer_ch8.clone(),
            mpsc_buffer_ch9: self.mpsc_buffer_ch9.clone(),
        }
    }
}

pub struct Mpscs {
    pub txs: MpscTxs,
    pub rxs: MpscRxs,
}

impl Mpscs {
    pub fn new() -> Self {
        let (tx_udp, rx_udp) = mpsc::channel(300);
        let (tx1, rx1) = mpsc::channel(300);
        let (tx2, rx2) = mpsc::channel(300);
        let (tx3, rx3) = mpsc::channel(300);
        let (tx4, rx4) = mpsc::channel(300);
        let (tx5, rx5) = mpsc::channel(300);

        // Initialisering av 10 nye buffer-kanalar
        let (tx_buf0, rx_buf0) = mpsc::channel(300);
        let (tx_buf1, rx_buf1) = mpsc::channel(300);
        let (tx_buf2, rx_buf2) = 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,
                tcp_to_master_failed: tx1,
                container: tx2,
                remove_container: tx3,
                local_elev: tx4,
                sent_tcp_container: tx5,

                // Legg til dei nye buffer-kanalane
                new_task: tx_buf0,
                update_task_status: tx_buf1,
                mpsc_buffer_ch2: tx_buf2,
                mpsc_buffer_ch3: tx_buf3,
                mpsc_buffer_ch4: tx_buf4,
                mpsc_buffer_ch5: 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,
                tcp_to_master_failed: rx1,
                container: rx2,
                remove_container: rx3,
                local_elev: rx4,
                sent_tcp_container: rx5,

                // Legg til dei nye buffer-kanalane
                new_task: rx_buf0,
                update_task_status: rx_buf1,
                mpsc_buffer_ch2: rx_buf2,
                mpsc_buffer_ch3: rx_buf3,
                mpsc_buffer_ch4: rx_buf4,
                mpsc_buffer_ch5: rx_buf5,
                mpsc_buffer_ch6: rx_buf6,
                mpsc_buffer_ch7: rx_buf7,
                mpsc_buffer_ch8: rx_buf8,
                mpsc_buffer_ch9: rx_buf9,
            },
        }
    }
}

impl Clone for Mpscs {
    fn clone(&self) -> Mpscs {
        let (_, rx_udp) = mpsc::channel(300);
        let (_, rx1) = mpsc::channel(300);
        let (_, rx2) = mpsc::channel(300);
        let (_, rx3) = mpsc::channel(300);
        let (_, rx4) = mpsc::channel(300);
        let (_, rx5) = mpsc::channel(300);

        // Initialiser mottakar-kanalane ved cloning
        let (_, rx_buf0) = mpsc::channel(300);
        let (_, rx_buf1) = mpsc::channel(300);
        let (_, rx_buf2) = mpsc::channel(300);
        let (_, rx_buf3) = mpsc::channel(300);
        let (_, rx_buf4) = mpsc::channel(300);
        let (_, rx_buf5) = mpsc::channel(300);
        let (_, rx_buf6) = mpsc::channel(300);
        let (_, rx_buf7) = mpsc::channel(300);
        let (_, rx_buf8) = mpsc::channel(300);
        let (_, rx_buf9) = mpsc::channel(300);

        Mpscs {
            txs: self.txs.clone(),
            rxs: MpscRxs {
                udp_wv: rx_udp,
                tcp_to_master_failed: rx1,
                container: rx2,
                remove_container: rx3,
                local_elev: rx4,
                sent_tcp_container: rx5,

                // Klonar buffer-kanalane
                new_task: rx_buf0,
                update_task_status: rx_buf1,
                mpsc_buffer_ch2: rx_buf2,
                mpsc_buffer_ch3: rx_buf3,
                mpsc_buffer_ch4: rx_buf4,
                mpsc_buffer_ch5: rx_buf5,
                mpsc_buffer_ch6: rx_buf6,
                mpsc_buffer_ch7: rx_buf7,
                mpsc_buffer_ch8: rx_buf8,
                mpsc_buffer_ch9: rx_buf9,
            },
        }
    }
}


// --- BROADCAST-KANALAR ---

pub struct BroadcastTxs {
    pub shutdown: broadcast::Sender<()>,
    pub broadcast_buffer_ch1: broadcast::Sender<bool>,
    pub broadcast_buffer_ch2: broadcast::Sender<bool>,
    pub broadcast_buffer_ch3: broadcast::Sender<bool>,
    pub broadcast_buffer_ch4: broadcast::Sender<bool>,
    pub broadcast_buffer_ch5: broadcast::Sender<bool>,
}

pub struct BroadcastRxs {
    pub shutdown: broadcast::Receiver<()>,
    pub broadcast_buffer_ch1: broadcast::Receiver<bool>,
    pub broadcast_buffer_ch2: broadcast::Receiver<bool>,
    pub broadcast_buffer_ch3: broadcast::Receiver<bool>,
    pub broadcast_buffer_ch4: broadcast::Receiver<bool>,
    pub broadcast_buffer_ch5: broadcast::Receiver<bool>,
}

impl Clone for BroadcastTxs {
    fn clone(&self) -> BroadcastTxs {
        BroadcastTxs {
            shutdown: self.shutdown.clone(),
            broadcast_buffer_ch1: self.broadcast_buffer_ch1.clone(),
            broadcast_buffer_ch2: self.broadcast_buffer_ch2.clone(),
            broadcast_buffer_ch3: self.broadcast_buffer_ch3.clone(),
            broadcast_buffer_ch4: self.broadcast_buffer_ch4.clone(),
            broadcast_buffer_ch5: self.broadcast_buffer_ch5.clone(),
        }
    }
}

impl BroadcastTxs {
    pub fn subscribe(&self) -> BroadcastRxs {
        BroadcastRxs {
            shutdown: self.shutdown.subscribe(),
            broadcast_buffer_ch1: self.broadcast_buffer_ch1.subscribe(),
            broadcast_buffer_ch2: self.broadcast_buffer_ch2.subscribe(),
            broadcast_buffer_ch3: self.broadcast_buffer_ch3.subscribe(),
            broadcast_buffer_ch4: self.broadcast_buffer_ch4.subscribe(),
            broadcast_buffer_ch5: self.broadcast_buffer_ch5.subscribe(),
        }
    }
}
```

```
impl BroadcastRxs {
    pub fn resubscribe(&self) -> BroadcastRxs {
        BroadcastRxs {
            shutdown: self.shutdown.resubscribe(),
            broadcast_buffer_ch1: self.broadcast_buffer_ch1.resubscribe(),
            broadcast_buffer_ch2: self.broadcast_buffer_ch2.resubscribe(),
            broadcast_buffer_ch3: self.broadcast_buffer_ch3.resubscribe(),
            broadcast_buffer_ch4: self.broadcast_buffer_ch4.resubscribe(),
            broadcast_buffer_ch5: self.broadcast_buffer_ch5.resubscribe(),
        }
    }
}

/// ## Structen inneholder alle Broadcast kanalene
///
/// Navn på kanalene er matchende for `txs` og `rxs`:
///
/// | Variabel  | Beskrivelse  |
/// |-----------|-------------|
/// | **shutdown**  | Signaliserer til alle tråder at de skal avslutte |
/// | **update_task_status**  | Buffer til fremtidig bruk |
/// | **mpsc_buffer_ch2**  | Buffer til fremtidig bruk |
/// | **mpsc_buffer_ch3**  | Buffer til fremtidig bruk |
/// | **local_elev**  | Buffer til fremtidig bruk |
/// | **sent_tcp_container**  | Buffer til fremtidig bruk |
pub struct Broadcasts {
    pub txs: BroadcastTxs,
    pub rxs: BroadcastRxs,
}

impl Broadcasts {
    pub fn new() -> Self {
        let (shutdown_tx, shutdown_rx) = broadcast::channel(1);
        let (tx1, rx1) = broadcast::channel(1);
        let (tx2, rx2) = broadcast::channel(1);
        let (tx3, rx3) = broadcast::channel(1);
        let (tx4, rx4) = broadcast::channel(1);
        let (tx5, rx5) = broadcast::channel(1);

        Broadcasts {
            txs: BroadcastTxs {
                shutdown: shutdown_tx,
                broadcast_buffer_ch1: tx1,
                broadcast_buffer_ch2: tx2,
                broadcast_buffer_ch3: tx3,
                broadcast_buffer_ch4: tx4,
                broadcast_buffer_ch5: tx5,
            },
            rxs: BroadcastRxs {
                shutdown: shutdown_rx,
                broadcast_buffer_ch1: rx1,
                broadcast_buffer_ch2: rx2,
                broadcast_buffer_ch3: rx3,
                broadcast_buffer_ch4: rx4,
                broadcast_buffer_ch5: rx5,
            },
        }
    }

    pub fn subscribe(&self) -> BroadcastRxs {
        self.txs.subscribe()
    }
}

impl Clone for Broadcasts {
    fn clone(&self) -> Broadcasts {
        Broadcasts {
            txs: self.txs.clone(),
            rxs: self.rxs.resubscribe(),
        }
    }
}

// --- WATCH-KANALER ---
pub struct WatchTxs {
    pub wv: watch::Sender<Vec<u8>>,
    pub elev_task: watch::Sender<Vec<Task>>,
    pub watch_buffer_ch2: watch::Sender<bool>,
    pub watch_buffer_ch3: watch::Sender<bool>,
    pub watch_buffer_ch4: watch::Sender<bool>,
    pub watch_buffer_ch5: watch::Sender<bool>,
}

impl Clone for WatchTxs {
    fn clone(&self) -> WatchTxs {
        WatchTxs {
            wv: self.wv.clone(),
            elev_task: self.elev_task.clone(),
            watch_buffer_ch2: self.watch_buffer_ch2.clone(),
            watch_buffer_ch3: self.watch_buffer_ch3.clone(),
            watch_buffer_ch4: self.watch_buffer_ch4.clone(),
            watch_buffer_ch5: self.watch_buffer_ch5.clone(),
        }
    }
}

pub struct WatchRxs {
    pub wv: watch::Receiver<Vec<u8>>,
    pub elev_task: watch::Receiver<Vec<Task>>,
    pub watch_buffer_ch2: watch::Receiver<bool>,
    pub watch_buffer_ch3: watch::Receiver<bool>,
    pub watch_buffer_ch4: watch::Receiver<bool>,
    pub watch_buffer_ch5: watch::Receiver<bool>,
}
```

```
impl Clone for WatchRxs {
    fn clone(&self) -> WatchRxs {
        WatchRxs {
            wv: self.wv.clone(),
            elev_task: self.elev_task.clone(),
            watch_buffer_ch2: self.watch_buffer_ch2.clone(),
            watch_buffer_ch3: self.watch_buffer_ch3.clone(),
            watch_buffer_ch4: self.watch_buffer_ch4.clone(),
            watch_buffer_ch5: self.watch_buffer_ch5.clone(),
        }
    }
}

/// ## Structen inneholder alle Watch kanalene
///
/// Navn på kanalene er matchende for `txs` og `rxs`:
///
/// | Variabel  | Beskrivelse  |
/// |-----------|-------------|
/// | **wv**  | wv oppdateres av ´world_view_handler´ og leses av i ´get_wv´ |
/// | **update_task_status**  | Buffer til fremtidig bruk |
/// | **mpsc_buffer_ch2**  | Buffer til fremtidig bruk |
/// | **mpsc_buffer_ch3**  | Buffer til fremtidig bruk |
/// | **local_elev**  | Buffer til fremtidig bruk |
/// | **sent_tcp_container**  | Buffer til fremtidig bruk |
pub struct Watches {
    pub txs: WatchTxs,
    pub rxs: WatchRxs,
}

impl Clone for Watches {
    fn clone(&self) -> Watches {
        Watches {
            txs: self.txs.clone(),
            rxs: self.rxs.clone(),
        }
    }
}

impl Watches {
    pub fn new() -> Self {
        let (wv_tx, wv_rx) = watch::channel(Vec::<u8>::new());
        let (tx1, rx1) = watch::channel(Vec::new());
        let (tx2, rx2) = watch::channel(false);
        let (tx3, rx3) = watch::channel(false);
        let (tx4, rx4) = watch::channel(false);
        let (tx5, rx5) = watch::channel(false);

        Watches {
            txs: WatchTxs {
                wv: wv_tx,
                elev_task: tx1,
                watch_buffer_ch2: tx2,
                watch_buffer_ch3: tx3,
                watch_buffer_ch4: tx4,
                watch_buffer_ch5: tx5,
            },
            rxs: WatchRxs {
                wv: wv_rx,
                elev_task: rx1,
                watch_buffer_ch2: rx2,
                watch_buffer_ch3: rx3,
                watch_buffer_ch4: rx4,
                watch_buffer_ch5: rx5,
            },
        }
    }
}

// --- SEMAPHORE-KANALAR ---

pub struct Semaphores {
    pub tcp_sent: Arc<Semaphore>,
    pub sem_buffer: Arc<Semaphore>,
}

impl Semaphores {
    pub fn new() -> Self {
        Semaphores {
            tcp_sent: Arc::new(Semaphore::new(10)),
            sem_buffer: Arc::new(Semaphore::new(5)),
        }
    }
}

impl Clone for Semaphores {
    fn clone(&self) -> Semaphores {
        Semaphores {
            tcp_sent: self.tcp_sent.clone(),
            sem_buffer: self.sem_buffer.clone(),
        }
    }
}


// --- OVERKLASSE FOR ALLE KANALAR ---


/// ## Overklasse for alle interne kanaler
///
/// Inneholder `MPSC`, `Broadcast` og `Watch` kanaler
pub struct LocalChannels {
    pub mpscs: Mpscs,
    pub broadcasts: Broadcasts,
    pub watches: Watches,
    pub semaphores: Semaphores,
}

impl LocalChannels {
    pub fn new() -> Self {
        LocalChannels {
            mpscs: Mpscs::new(),
            broadcasts: Broadcasts::new(),
            watches: Watches::new(),
            semaphores: Semaphores::new(),
        }
    }

    pub fn subscribe_broadcast(&mut self) {
        self.broadcasts.rxs = self.broadcasts.subscribe();
    }

    pub fn resubscribe_broadcast(&mut self) {
        self.broadcasts.rxs = self.broadcasts.rxs.resubscribe();
    }
}

impl Clone for LocalChannels {
    fn clone(&self) -> LocalChannels {
        LocalChannels {
            mpscs: self.mpscs.clone(),
            broadcasts: self.broadcasts.clone(),
            watches: self.watches.clone(),
            semaphores: self.semaphores.clone(),
        }
    }
}
```

Innhald frå Rust-filer

Fil: elevator_pro\src\network\tcp_network.rs

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

use std::sync::atomic::{AtomicBool, Ordering};
use tokio::{io::{AsyncReadExt, AsyncWriteExt}, net::{TcpListener, TcpStream}, task::JoinHandle, sync::mpsc,
time::{sleep, Duration, Instant}};
use std::net::SocketAddr;
use crate::{config, utils::{self, SELF_ID, print_info, print_ok, print_err, get_wv, update_wv},
world_view::{world_view_update, world_view}};
use super::local_network;

// Definer ein global `AtomicU8`
pub static IS_MASTER: AtomicBool = AtomicBool::new(false); // Startverdi 0

/// ### TcpWatchdog
///
/// Håndterer timeout på TCP connections hos master, og lesing fra slave
struct TcpWatchdog {
    timeout: Duration,
}

impl TcpWatchdog {
    /// Starter en asynkron løkke der vi veksler mellom å lese fra stream og sjekke for timeout.
    async fn start_reading_from_slave(&self, mut stream: TcpStream, chs: local_network::LocalChannels) {
        let mut last_success = Instant::now();

        loop {
            // Kalkulerer hvor lang tid vi har igjen før timeout inntreffer.
            let remaining = self.timeout
                .checked_sub(last_success.elapsed())
                .unwrap_or(Duration::from_secs(0));

            // Lager en sleep-future basert på gjenværende tid.
            let sleep_fut = sleep(remaining);
            tokio::pin!(sleep_fut);

            tokio::select! {
                // Forsøker å lese fra stream med de nødvendige parameterne.
                result = read_from_stream(&mut stream, chs.clone()) => {
                    match result {
                        Some(msg) => {
                            let _ = chs.mpscs.txs.container.send(msg).await;
                            last_success = Instant::now()
                        }
                        None => {
                            break;
                        }
                    }
                }
                // Triggeres dersom ingen melding er mottatt innen timeouttiden.
                _ = &mut sleep_fut => {
                    utils::print_err(format!("Timeout: Ingen melding mottatt innen {:?}", self.timeout));
                    let id = utils::ip2id(stream.peer_addr().expect("Peer har ingen IP?").ip());
                    utils::print_info(format!("Stenger stream til slave {}", id));
                    let _ = chs.mpscs.txs.remove_container.send(id).await;
                    let _ = stream.shutdown().await;
                    break;
                }
            }
        }
    }
}



/// ### Håndterer TCP-connections
pub async fn tcp_handler(chs: local_network::LocalChannels, mut socket_rx: mpsc::Receiver<(TcpStream, SocketAddr)>) {
    let mut wv = get_wv(chs.clone());
    loop {
        IS_MASTER.store(true, Ordering::SeqCst);
        /* Mens du er master: Motta sockets til slaver, start handle_slave i ny task*/
        while utils::is_master(wv.clone()) {
            if world_view_update::get_network_status().load(Ordering::SeqCst) {
                while let Ok((socket, addr)) = socket_rx.try_recv() {
                    let chs_clone = chs.clone();
                    utils::print_info(format!("Ny slave tilkobla: {}", addr));
                    let _slave_task: JoinHandle<()> = tokio::spawn(async move {
                        let tcp_watchdog = TcpWatchdog {
                            timeout: Duration::from_millis(config::TCP_TIMEOUT),
                        };
                        // Starter watchdogløkken, håndterer også mottak av meldinger på socketen
                        tcp_watchdog.start_reading_from_slave(socket, chs_clone).await;
                    });
                    tokio::task::yield_now().await; //Denne tvinger tokio til å sørge for at alle tasks i kø blir behandler
                                                            //Feilen før var at tasken ble lagd i en loop, og try_recv kaltes så tett att tokio ikke rakk å starte tasken før man fikk en ny melding(og den fikk litt tid da den mottok noe)
                }
            }
            else {
                tokio::time::sleep(Duration::from_millis(100)).await;
            }
            update_wv(chs.clone(), &mut wv).await;
        }
        //mista master -> indiker for avslutning av tcp-con og tasks
        IS_MASTER.store(false, Ordering::SeqCst);


        // sjekker at vi faktisk har ein socket å bruke med masteren
        let mut master_accepted_tcp = false;
        let mut stream:Option<TcpStream> = None;
        if let Some(s) = connect_to_master(chs.clone()).await {
            println!("Master accepta tilkobling");
            master_accepted_tcp = true;
            stream = Some(s);
        } else {
            println!("Master accepta IKKE tilkobling");
        }

        /* Mens du er slave: Sjekk om det har kommet ny master / connection til master har dødd */
        let mut prev_master: u8;
        let mut new_master = false;
        while !utils::is_master(wv.clone()) && master_accepted_tcp {

            if world_view_update::get_network_status().load(Ordering::SeqCst) {
                if let Some(ref mut s) = stream {
                    if new_master {
                        utils::print_slave(format!("Fått ny master"));
                        master_accepted_tcp = false;
                        utils::slave_sleep().await;
                    }
                    prev_master = wv[config::MASTER_IDX];
                    update_wv(chs.clone(), &mut wv).await;
                    // Send neste TCP melding til master
                    send_tcp_message(chs.clone(), s, wv.clone()).await;
                    if prev_master != wv[config::MASTER_IDX] {
                        new_master = true;
                    }
                    tokio::time::sleep(config::TCP_PERIOD).await;
                }
            }
            else {
                utils::slave_sleep().await;
            }
        }
        //ble master -> restart loopen
    }
}

/// ### Forsøker å koble til master via TCP.
/// Returnerer `Some(TcpStream)` ved suksess, `None` ved feil.
async fn connect_to_master(chs: local_network::LocalChannels) -> Option<TcpStream> {
    let wv = get_wv(chs.clone());

    // Sjekker at vi har internett før vi prøver å koble til
    if world_view_update::get_network_status().load(Ordering::SeqCst) {
        let master_ip = format!("{}.{}:{}", config::NETWORK_PREFIX, wv[config::MASTER_IDX], config::PN_PORT);
        print_info(format!("Prøver å koble på: {} i TCP_listener()", master_ip));

        // Prøv å koble til master
        match TcpStream::connect(&master_ip).await {
            Ok(stream) => {
                print_ok(format!("Har kobla på Master: {} i TCP_listener()", master_ip));
                // Klarte å koble til master, returner streamen
                Some(stream)
            }
            Err(e) => {
                print_err(format!("Klarte ikke koble på master tcp: {}", e));

                match chs.mpscs.txs.tcp_to_master_failed.send(true).await {
                    Ok(_) => print_info("Sa ifra at TCP til master feila".to_string()),
                    Err(err) => print_err(format!("Feil ved sending til tcp_to_master_failed: {}", err)),
                }
                None
            }
        }
    } else {
        None
    }
}

/// ### Starter og kjører TCP-listener
pub async fn listener_task(_chs: local_network::LocalChannels, socket_tx: mpsc::Sender<(TcpStream, SocketAddr)>) {
    let self_ip = format!("{}.{}", config::NETWORK_PREFIX, SELF_ID.load(Ordering::SeqCst));
    // Ved første init, vent til vi er sikre på at vi har internett
    while !world_view_update::get_network_status().load(Ordering::SeqCst) {
        tokio::time::sleep(config::TCP_PERIOD).await;
    }

    /* Binder listener til PN_PORT */
    let listener = match TcpListener::bind(format!("{}:{}", self_ip, config::PN_PORT)).await {
        Ok(l) => {
            utils::print_ok(format!("Master lytter på {}:{}", self_ip, config::PN_PORT));
            l
        }
        Err(e) => {
            utils::print_err(format!("Feil ved oppstart av TCP-listener: {}", e));
            return; // evt gå i sigel elevator mode
        }
    };

    /* Når listener accepter ny tilkobling -> send socket og addr til tcp_handler gjennom socket_tx */
    loop {
        sleep(Duration::from_millis(100)).await;
        match listener.accept().await {
            Ok((socket, addr)) => {
                utils::print_master(format!("{} kobla på TCP", addr));
                if socket_tx.send((socket, addr)).await.is_err() {
                    utils::print_err("Hovudløkken har stengt, avsluttar listener.".to_string());
                    break;
                }
            }
            Err(e) => {
                utils::print_err(format!("Feil ved tilkobling av slave: {}", e));
            }
        }
    }
}
```

Innhald frå Rust-filer

```
/// ## Leser fra `stream`
///
/// Select mellom å lese melding fra slave og sende meldingen til `world_view_handler` og å avslutte streamen om du ikke er master
async fn read_from_stream(stream: &mut TcpStream, chs: local_network::LocalChannels) -> Option<Vec<u8>> {
    let mut len_buf = [0u8; 2];
    tokio::select! {
        result = stream.read_exact(&mut len_buf) => {
            match result {
                Ok(0) => {
                    utils::print_info("Slave har kopla fra.".to_string());
                    utils::print_info(format!("Stenger stream til slave 1: {:?}", stream.peer_addr()));
                    let id = utils::ip2id(stream.peer_addr().expect("Peer har ingen IP?").ip());
                    let _ = chs.mpscs.txs.remove_container.send(id).await;
                    // let _ = stream.shutdown().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) => {
                            utils::print_info("Slave har kopla fra.".to_string());
                            utils::print_info(format!("Stenger stream til slave 2: {:?}", stream.peer_addr()));
                            let id = utils::ip2id(stream.peer_addr().expect("Peer har ingen IP?").ip());
                            let _ = chs.mpscs.txs.remove_container.send(id).await;
                            // let _ = stream.shutdown().await;
                            return None;
                        }
                        Ok(_) => return Some(buffer),
                        Err(e) => {
                            utils::print_err(format!("Feil ved mottak av data fra slave: {}", e));
                            utils::print_info(format!("Stenger stream til slave 3: {:?}", stream.peer_addr()));
                            let id = utils::ip2id(stream.peer_addr().expect("Peer har ingen IP?").ip());
                            let _ = chs.mpscs.txs.remove_container.send(id).await;
                            // let _ = stream.shutdown().await;
                            return None;
                        }
                    }
                }
                Err(e) => {
                    utils::print_err(format!("Feil ved mottak av data fra slave: {}", e));
                    utils::print_info(format!("Stenger stream til slave 4: {:?}", stream.peer_addr()));
                    let id = utils::ip2id(stream.peer_addr().expect("Peer har ingen IP?").ip());
                    let _ = chs.mpscs.txs.remove_container.send(id).await;
                    // let _ = stream.shutdown().await;
                    return None;
                }
            }
        }
        _ = async {
            while IS_MASTER.load(Ordering::SeqCst) {
                tokio::time::sleep(Duration::from_millis(50)).await;
            }
        } => {
            let id = utils::ip2id(stream.peer_addr().expect("Peer har ingen IP?").ip());
            utils::print_info(format!("Mistar masterstatus, stenger stream til slave {}", id));
            let _ = chs.mpscs.txs.remove_container.send(id).await;
            // let _ = stream.shutdown().await;
            return None;
        }
    }
}

/// ### Sender egen elevator_container til master gjennom stream
/// Sender på format : `(lengde av container) as u16`, `container`
pub async fn send_tcp_message(chs: local_network::LocalChannels, stream: &mut TcpStream, wv: Vec<u8>) {
    let self_elev_container = utils::extract_self_elevator_container(wv);

    let self_elev_serialized = world_view::serialize_elev_container(&self_elev_container);
    let len = (self_elev_serialized.len() as u16).to_be_bytes(); // Konverter lengde til big-endian bytes

    if let Err(e) = stream.write_all(&len).await {
        // utils::print_err(format!("Feil ved sending av data til master: {}", e));
        let _ = chs.mpscs.txs.tcp_to_master_failed.send(true).await; // Anta at tilkoblingen feila
    } else if let Err(e) = stream.write_all(&self_elev_serialized).await {

        // utils::print_err(format!("Feil ved sending av data til master: {}", e));
        let _ = chs.mpscs.txs.tcp_to_master_failed.send(true).await; // Anta at tilkoblingen feila
    } else if let Err(e) = stream.flush().await {
        // utils::print_err(format!("Feil ved flushing av stream: {}", e));
        let _ = chs.mpscs.txs.tcp_to_master_failed.send(true).await; // Anta at tilkoblingen feila
    } else {
        // send_succes_l = true;
        let _ = chs.mpscs.txs.sent_tcp_container.send(self_elev_serialized).await;
    }
}
```

Innhald frå Rust-filer

Fil: elevator_pro\src\network\tcp_self_elevator.rs

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

use crate::elevator_logic::task_handler;
use crate::utils::SELF_ID;
use crate::world_view::world_view;
use crate::{config, utils::{self, print_ok}, world_view::world_view_update, elevio, elevio::poll::CallButton, elevio::elev as
e};

use super::local_network;



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

struct LocalElevRxs {
    call_button: cbc::Receiver<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::poll::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 }
        }
    }
}
```

Innhald frå Rust-filer

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

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

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

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

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


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

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

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

/// ### Kjører den lokale heisen
pub async fn run_local_elevator(chs: local_network::LocalChannels) -> std::io::Result<()> {
    // Start elevator-serveren
    start_elevator_server().await;
    let local_elev_channels: LocalElevChannels = LocalElevChannels::new();
    utils::slave_sleep().await;
                                    let elevator: e::Elevator = e::Elevator::init(config::LOCAL_ELEV_IP,
config::DEFAULT_NUM_FLOORS).expect("Feil!");

    // Start polling på meldinger fra heisen
    {
        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)
        });
    }

    //Start en task som viderefører meldinger fra heisen til update_worldview
    {
        let chs_clone = chs.clone();
        let _listen_task = tokio::spawn(async move {
            let _ = read_from_local_elevator(local_elev_channels.rxs, chs_clone).await;
        });
    }

    // Task som utfører deligerte tasks (ikke implementert korrekt enda)
    {
        let chs_clone = chs.clone();
        let _handle_task = tokio::spawn(async move {
            let _ = task_handler::execute_tasks(chs_clone, elevator).await;
        });
        tokio::task::yield_now().await;
    }

    // Loop som sender egen container på kanalen som motar slave-kontainere hvis man er master
    let mut wv = utils::get_wv(chs.clone());
    loop {
        utils::update_wv(chs.clone(), &mut wv).await;
        if utils::is_master(wv.clone()) {
            /* Oppdater task og task_status, send din container tilbake som om den fikk fra tcp */
            let wv_deser = world_view::deserialize_worldview(&world_view_update::join_wv(wv.clone(), wv.clone()));
```

```
                let self_idx = world_view::get_index_to_container(SELF_ID.load(Ordering::SeqCst),
world_view::serialize_worldview(&wv_deser));
            if let Some(i) = self_idx {
                let _ =
chs.mpscs.txs.container.send(world_view::serialize_elev_container(&wv_deser.elevator_containers[i])).await;
            }
        }
        sleep(config::TCP_PERIOD).await;
    }
}

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

        if let Ok(floor) = rxs.floor_sensor.try_recv() {
            //println!("Floor: {:#?}", floor);
            let msg = local_network::ElevMessage {
                msg_type: local_network::ElevMsgType::FSENS,
                call_button: None,
                floor_sensor: Some(floor),
                stop_button: None,
                obstruction: None,
            };
            let _ = chs.mpscs.txs.local_elev.send(msg).await;
        }

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

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

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

Innhald frå Rust-filer

Fil: elevator_pro\src\network\udp_broadcast.rs

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

use crate::config;
use crate::utils;
use super::local_network;

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 std::borrow::Cow;

static UDP_TIMEOUT: OnceLock<AtomicBool> = OnceLock::new(); // worldview_channel_request
pub fn get_udp_timeout() -> &'static AtomicBool {
    UDP_TIMEOUT.get_or_init(|| AtomicBool::new(false))
}

/// ### Starter og kjører udp-broadcaster
pub async fn start_udp_broadcaster(mut chs: local_network::LocalChannels) -> tokio::io::Result<()> {
    // Sett opp sockets
    chs.subscribe_broadcast();
    let addr: &str = &format!("{}:{}", config::BC_ADDR, config::DUMMY_PORT);
    let addr2: &str = &format!("{}:0", config::BC_LISTEN_ADDR);

    let broadcast_addr: SocketAddr = addr.parse().expect("ugyldig adresse"); // UDP-broadcast adresse
    let socket_addr: SocketAddr = addr2.parse().expect("Ugyldig adresse");
    let socket = Socket::new(Domain::IPV4, Type::DGRAM, None)?;


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

    let mut wv = utils::get_wv(chs.clone());
    loop{
        let chs_clone = chs.clone();
        utils::update_wv(chs_clone, &mut wv).await;

        // Hvis du er master, broadcast worldview
        if utils::SELF_ID.load(Ordering::SeqCst) == wv[config::MASTER_IDX] {
            //TODO: Lag bedre delay?
            sleep(config::UDP_PERIOD);
            let mesage = format!("{:?}{:?}", config::KEY_STR, wv).to_string();
            udp_socket.send_to(mesage.as_bytes(), &broadcast_addr).await?;
        }
    }
}

/// ### Starter og kjører udp-listener
pub async fn start_udp_listener(mut chs: local_network::LocalChannels) -> tokio::io::Result<()> {
    //Sett opp sockets
    chs.subscribe_broadcast();
    let self_id = utils::SELF_ID.load(Ordering::SeqCst);
    let broadcast_listen_addr = format!("{}:{}", config::BC_LISTEN_ADDR, config::DUMMY_PORT);
    let socket_addr: SocketAddr = broadcast_listen_addr.parse().expect("Ugyldig adresse");
    let socket_temp = Socket::new(Domain::IPV4, Type::DGRAM, None)?;



    socket_temp.set_reuse_address(true)?;
    socket_temp.set_broadcast(true)?;
    socket_temp.bind(&socket_addr.into())?;
    let socket = UdpSocket::from_std(socket_temp.into())?;
    let mut buf = [0; config::UDP_BUFFER];
    let mut read_wv: Vec<u8> = Vec::new();

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

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

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

        }
    }
}


/// ### jalla udp watchdog
pub async fn udp_watchdog(chs: local_network::LocalChannels) {
    loop {
        if get_udp_timeout().load(Ordering::SeqCst) == false {
            get_udp_timeout().store(true, Ordering::SeqCst);
            tokio::time::sleep(Duration::from_millis(1000)).await;
        }
        else {
            get_udp_timeout().store(false, Ordering::SeqCst); //resetter watchdogen
            utils::print_warn("UDP-watchdog: Timeout".to_string());
            let _ = chs.mpscs.txs.tcp_to_master_failed.send(true).await;
        }
    }
}
```

Innhald frå Rust-filer

Fil: elevator_pro\src\world_view\world_view.rs

```
use serde::{Serialize, Deserialize};
use crate::config;
use crate::utils;
use crate::elevio::poll::CallType;
use ansi_term::Colour::{Blue, Green, Red, Yellow, Purple};
use ansi_term::Style;
use prettytable::{Table, Row, Cell, format, Attr, color};
use crate::elevio::poll::CallButton;



#[derive(Serialize, Deserialize, Debug, Default, Clone, Hash)]
pub struct Task {
    pub id: u16,
    pub to_do: u8, // Default: 0
    pub status: TaskStatus, // 1: done, 0: to_do, 255: be master deligere denne på nytt
    pub is_inside: bool,
}

#[derive(Serialize, Deserialize, Debug, Clone, PartialEq, Hash)]
pub enum TaskStatus {
    PENDING,
    DONE,
    UNABLE = u8::MAX as isize,
}
impl Default for TaskStatus {
    fn default() -> Self {
        TaskStatus::PENDING
    }
}


#[derive(Serialize, Deserialize, Debug, Clone)]
pub struct ElevatorContainer {
    pub elevator_id: u8,            // Default: 0
    pub calls: Vec<CallButton>,     // Default: vektor med Tasks
    pub tasks: Vec<Task>,           // Default: vektor med Tasks
    pub tasks_status: Vec<Task>,    // Default: vektor med Tasks Slave skriver, Master leser
    pub door_open: bool,            // Default: false
    pub obstruction: bool,          // Default: false
    pub motor_dir: u8,              // Default: 0
    pub last_floor_sensor: u8,      // Default: 255
}
impl Default for ElevatorContainer {
    fn default() -> Self {
        Self {
            elevator_id: 0,
            calls: Vec::new(),
            tasks: Vec::new(),
            tasks_status: Vec::new(),
            door_open: false,
            obstruction: false,
            motor_dir: 0,
            last_floor_sensor: 255, // Spesifikk verdi for sensor
        }
    }
}


#[derive(Serialize, Deserialize, Debug, Clone)]
pub struct WorldView {
    //Generelt nettverk
    n: u8,                          // Antall heiser
    pub master_id: u8,                  // Master IP
    //Generelle oppgaver til heisen
    pub outside_button: Vec<CallButton>,          // Array til knappene trykt på utsiden
    //Heisspesifikt
    pub elevator_containers: Vec<ElevatorContainer>,   //Info som gjelder per-heis

}


impl Default for WorldView {
     fn default() -> Self {
        Self {
            n: 0,
            master_id: config::ERROR_ID,
            outside_button: Vec::new(),
            elevator_containers: Vec::new(),
        }
    }
}


impl WorldView {
    pub fn add_elev(&mut self, elevator: ElevatorContainer) {
        // utils::print_ok(format!("elevator med ID {} ble ansatt. (add_elev())", elevator.elevator_id));
        self.elevator_containers.push(elevator);
        self.n = self.elevator_containers.len() as u8;
    }

    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 {
            utils::print_warn(format!("Ingen elevator med ID {} ble funnet. (remove_elev())", id));
        } else {
            utils::print_ok(format!("elevator med ID {} ble sparka. (remove_elev())", id));
        }
        self.n = self.elevator_containers.len() as u8;
    }

    pub fn get_num_elev(&self) -> u8 {
        return self.n;
    }

    pub fn set_num_elev(&mut self, n: u8)  {
        self.n = n;
    }
}




pub fn serialize_worldview(worldview: &WorldView) -> Vec<u8> {
    let encoded = bincode::serialize(worldview);
    match encoded {
        Ok(serialized_data) => {
            // Deserialisere WorldView fra binært format
            return serialized_data;
        }
        Err(e) => {
            println!("{:?}", worldview);
            utils::print_err(format!("Serialization failed: {} (world_view.rs, serialize_worldview())", e));
            panic!();
        }
    }
}

// Funksjon for å deserialisere WorldView
pub fn deserialize_worldview(data: &[u8]) -> WorldView {
    let decoded = bincode::deserialize(data);


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


pub fn serialize_elev_container(elev_container: &ElevatorContainer) -> Vec<u8> {
    let encoded = bincode::serialize(elev_container);
    match encoded {
        Ok(serialized_data) => {
            // Deserialisere WorldView fra binært format
            return serialized_data;
        }
        Err(e) => {
            utils::print_err(format!("Serialization failed: {} (world_view.rs, serialize_elev_container())", e));
            panic!();
        }
    }
}

// Funksjon for å deserialisere WorldView
pub fn deserialize_elev_container(data: &[u8]) -> ElevatorContainer {
    let decoded = bincode::deserialize(data);


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

pub fn get_index_to_container(id: u8, wv: Vec<u8>) -> Option<usize> {
    let wv_deser = deserialize_worldview(&wv);
    for i in 0..wv_deser.get_num_elev() {
        if wv_deser.elevator_containers[i as usize].elevator_id == id {
            return Some(i as usize);
        }
    }
    return None;
}


/// ### Printer wv på et pent og oversiktlig format
pub fn print_wv(worldview: Vec<u8>) {
    let mut print_stat = true;
    unsafe {
        print_stat = config::PRINT_WV_ON;
    }
    if !print_stat {
        return;
    }

    let wv_deser = deserialize_worldview(&worldview);
    let mut gen_table = Table::new();
    gen_table.set_format(*format::consts::FORMAT_CLEAN);
    let mut table = Table::new();
    table.set_format(*format::consts::FORMAT_CLEAN);
```

Innhald frå Rust-filer

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

//Legg til generell worldview-info
//Funka ikke når jeg brukte fargene på lik måte som under. gudene vet hvorfor
gen_table.add_row(Row::new(vec![
    Cell::new("Num heiser").with_style(Attr::ForegroundColor(color::BRIGHT_BLUE)),
    Cell::new("MasterID").with_style(Attr::ForegroundColor(color::BRIGHT_BLUE)),
    Cell::new("Outside Buttons").with_style(Attr::ForegroundColor(color::BRIGHT_BLUE)),
]));

let n_text = format!("{}", wv_deser.get_num_elev()); // Fjern ANSI og bruk prettytable farge
let m_id_text = format!("{}", wv_deser.master_id);
let button_list = wv_deser.outside_button.iter()
.map(|c| match c.call {
    CallType::INSIDE => format!("{}:{:?}({})", c.floor, c.call, c.elev_id),
    _ => format!("{}:{:?}:PUBLIC", c.floor, c.call),
})
.collect::<Vec<String>>()
.join(", ");

gen_table.add_row(Row::new(vec![
    Cell::new(&n_text).with_style(Attr::ForegroundColor(color::BRIGHT_YELLOW)),
    Cell::new(&m_id_text).with_style(Attr::ForegroundColor(color::BRIGHT_YELLOW)),
    Cell::new(&button_list),
]));

gen_table.printstd();



// Legg til heis-spesifikke deler
// Legg til hovudrad (header) med blå feittskrift
table.add_row(Row::new(vec![
    Cell::new(&Blue.bold().paint("ID").to_string()),
    Cell::new(&Blue.bold().paint("Dør").to_string()),
    Cell::new(&Blue.bold().paint("Obstruksjon").to_string()),
    Cell::new(&Blue.bold().paint("Motor Retning").to_string()),
    Cell::new(&Blue.bold().paint("Siste etasje").to_string()),
    Cell::new(&Blue.bold().paint("Tasks (ToDo:Status)").to_string()),
    Cell::new(&Blue.bold().paint("Calls (Etg:Call)").to_string()),
    Cell::new(&Blue.bold().paint("Tasks_status (ToDo:Status)").to_string()),
]));

// Iterer over alle heisane
for elev in wv_deser.elevator_containers {
    // Lag ein fargerik streng for ID
    let id_text = Yellow.bold().paint(format!("{}", elev.elevator_id)).to_string();

    // Door og obstruction i grøn/raud
    let door_status = if elev.door_open {
```

```
    Yellow.paint("Åpen").to_string()
} else {
    Green.paint("Lukket").to_string()
};

let obstruction_status = if elev.obstruction {
    Red.paint("Ja").to_string()
} else {
    Green.paint("Nei").to_string()
};

let task_color = match elev.tasks.len() {
    0..=1 => Green,  // Få oppgåver
    2..=4 => Yellow, // Middels mange oppgåver
    _ => Red, // Mange oppgåver
};
// Farge basert på `to_do`
let task_list = elev.tasks.iter()
    .map(|t| {
        format!("{}:{}:{}",
        task_color.paint(t.id.to_string()),
        task_color.paint(t.to_do.to_string()),
            task_color.paint(format!("{:?}", t.status))
        )
    })
    .collect::<Vec<String>>()
    .join(", ");

// Vanleg utskrift av calls
let call_list = elev.calls.iter()
    .map(|c| format!("{}:{:?}", c.floor, c.call))
    .collect::<Vec<String>>()
    .join(", ");

let task_stat_list = elev.tasks_status.iter()
    .map(|t| {
        format!("{}:{}:{}",
        task_color.paint(t.id.to_string()),
        task_color.paint(t.to_do.to_string()),
            task_color.paint(format!("{:?}", t.status))
        )
    })
    .collect::<Vec<String>>()
    .join(", ");

table.add_row(Row::new(vec![
    Cell::new(&id_text),
    Cell::new(&door_status),
    Cell::new(&obstruction_status),
    Cell::new(&format!("{}", elev.motor_dir)),
    Cell::new(&format!("{}", elev.last_floor_sensor)),
    Cell::new(&task_list),
            Cell::new(&call_list),
            Cell::new(&task_stat_list),
        ]));
    }

    // Skriv ut tabellen med fargar (ANSI-kodar)
    table.printstd();
    print!("\n\n");
}
```

Innhald frå Rust-filer

Fil: elevator_pro\src\world_view\world_view_ch.rs

```
use std::sync::atomic::Ordering;
use std::u16;
use tokio::time::sleep;

use crate::config;
use crate::elevio::poll::CallButton;
use crate::world_view::world_view;
use crate::world_view::world_view::TaskStatus;
use crate::network::tcp_network;
use crate::world_view::world_view_update;
use crate::network::local_network::{self, ElevMessage};
use std::collections::HashSet;
use crate::utils::{self, print_err, print_info, print_ok};
use crate::elevator_logic::master;

use super::world_view::Task;



/// ### Oppdatering av lokal worldview
///
/// Funksjonen leser nye meldinger fra andre tasks som indikerer endring i systemet, og endrer og oppdaterer det lokale worldviewen basert på dette.
pub async fn update_wv(mut main_local_chs: local_network::LocalChannels, mut worldview_serialised: Vec<u8>) {
    println!("Starter update_wv");
    let _ = main_local_chs.watches.txs.wv.send(worldview_serialised.clone());

    let mut wv_edited_I = false;
    loop {
        //OBS: Error kommer når kanal er tom. ikke print der uten å eksplisitt ekskludere channel_empty error type

/* KANALER SLAVE HOVEDSAKLIG MOTTAR PÅ */
        /*_____Fjerne knappar som vart sendt på TCP_____ */
        match main_local_chs.mpscs.rxs.sent_tcp_container.try_recv() {
            Ok(msg) => {
                wv_edited_I = clear_from_sent_tcp(&mut worldview_serialised, msg);
            },
            Err(_) => {},
        }
        /*_____Oppdater WV fra UDP-melding_____ */
        match main_local_chs.mpscs.rxs.udp_wv.try_recv() {
            Ok(master_wv) => {
                wv_edited_I = join_wv_from_udp(&mut worldview_serialised, master_wv);
            },
            Err(_) => {},
        }
        /*_____Signal om at tilkobling til master har feila_____ */
        match main_local_chs.mpscs.rxs.tcp_to_master_failed.try_recv() {
            Ok(_) => {
                wv_edited_I = abort_network(&mut worldview_serialised);
            },
      Err(_) => {},
    }


/* KANALER MASTER HOVEDSAKLIG MOTTAR PÅ */
    /*_____Melding til master fra slaven (elevator-containeren til slaven)_____*/
    match main_local_chs.mpscs.rxs.container.try_recv() {
      Ok(container) => {
        wv_edited_I = join_wv_from_tcp_container(&mut worldview_serialised, container).await;
      },
      Err(_) => {},
    }
    /*_____ID til slave som er død (ikke kontakt med slave)_____ */
    match main_local_chs.mpscs.rxs.remove_container.try_recv() {
      Ok(id) => {
        wv_edited_I = remove_container(&mut worldview_serialised, id);
      },
      Err(_) => {},
    }
    match main_local_chs.mpscs.rxs.new_task.try_recv() {
      Ok((task ,id, button)) => {
        // utils::print_master(format!("Fikk task: {:?}", task));
        wv_edited_I = push_task(&mut worldview_serialised, task, id, button);
      },
      Err(_) => {},
    }



/* KANALER MASTER OG SLAVE MOTTAR PÅ */
    /*_____Knapper trykket på lokal heis_____ */
    match main_local_chs.mpscs.rxs.local_elev.try_recv() {
      Ok(msg) => {
        wv_edited_I = recieve_local_elevator_msg(&mut worldview_serialised, msg).await;
      },
      Err(_) => {},
    }
    /*____Får signal når en task er ferdig_____ */
    match main_local_chs.mpscs.rxs.update_task_status.try_recv() {
      Ok((id, status)) => {
        println!("Skal sette status {:?} på task id: {}", status, id);
        wv_edited_I = update_task_status(&mut worldview_serialised, id, status);
      },
      Err(_) => {},
    }



/* KANALER ALLE SENDER LOKAL WV PÅ */
    /*_____Hvis worldview er endra, oppdater kanalen_____ */
    if wv_edited_I {
      let _ = main_local_chs.watches.txs.wv.send(worldview_serialised.clone());
```

```
            // println!("Sendte worldview lokalt {}", worldview_serialised[1]);

            wv_edited_I = false;
        }
    }
}

/// ### Oppdater WorldView fra master sin UDP melding
pub fn join_wv_from_udp(wv: &mut Vec<u8>, master_wv: Vec<u8>) -> bool {
    *wv = world_view_update::join_wv(wv.clone(), master_wv);
    true
}

/// ### 'Forlater' nettverket, fjerner alle heiser som ikke er seg selv
pub fn abort_network(wv: &mut Vec<u8>) -> bool {
    let mut deserialized_wv = world_view::deserialize_worldview(wv);
    deserialized_wv.elevator_containers.retain(|elevator| elevator.elevator_id == utils::SELF_ID.load(Ordering::SeqCst));
    deserialized_wv.set_num_elev(deserialized_wv.elevator_containers.len() as u8);
    deserialized_wv.master_id = utils::SELF_ID.load(Ordering::SeqCst);
    *wv = world_view::serialize_worldview(&deserialized_wv);
    true
}

/// ### Oppdaterer worldview basert på TCP melding fra slave
pub async fn join_wv_from_tcp_container(wv: &mut Vec<u8>, container: Vec<u8>) -> bool {
    let deser_container = world_view::deserialize_elev_container(&container);
    let mut deserialized_wv = world_view::deserialize_worldview(&wv);

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

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

    if let Some(i) = self_idx {
        //Oppdater statuser + fjerner tasks som er TaskStatus::DONE
        master::wv_from_slaves::update_statuses(&mut deserialized_wv, &deser_container, i).await;
        //Oppdater call_buttons
        master::wv_from_slaves::update_call_buttons(&mut deserialized_wv, &deser_container, i).await;
        *wv = world_view::serialize_worldview(&deserialized_wv);
        return true;
    } else {
         //Hvis dette printes, finnes ikke slaven i worldview. I teorien umulig, ettersom slaven blir lagt til over hvis den ikke
allerede eksisterte
        utils::print_cosmic_err("The elevator does not exist join_wv_from_tcp_conatiner()".to_string());
        return false;
    }
}

/// ### Fjerner slave basert på ID
```

Innhald frå Rust-filer

```
pub fn remove_container(wv: &mut Vec<u8>, id: u8) -> bool {
    let mut deserialized_wv = world_view::deserialize_worldview(&wv);
    deserialized_wv.remove_elev(id);
    *wv = world_view::serialize_worldview(&deserialized_wv);
    true
}

/// ### Behandler meldinger fra egen heis
pub async fn recieve_local_elevator_msg(wv: &mut Vec<u8>, msg: ElevMessage) -> bool {
    let is_master = utils::is_master(wv.clone());
    let mut deserialized_wv = world_view::deserialize_worldview(&wv);
    let self_idx = world_view::get_index_to_container(utils::SELF_ID.load(Ordering::SeqCst) , wv.clone());

    // Matcher hvilken knapp-type som er mottat
    match msg.msg_type {
        // Callbutton -> Legg den til i calls under egen heis-container
        local_network::ElevMsgType::CBTN => {
            print_info(format!("Callbutton: {:?}", msg.call_button));
            if let (Some(i), Some(call_btn)) = (self_idx, msg.call_button) {
                deserialized_wv.elevator_containers[i].calls.push(call_btn);

                //Om du er master i nettverket, oppdater call_buttons (Samme funksjon som kjøres i join_wv_from_tcp_container(). Behandler altså egen heis som en slave i nettverket)
                if is_master {
                    let container = deserialized_wv.elevator_containers[i].clone();
                    master::wv_from_slaves::update_call_buttons(&mut deserialized_wv, &container, i).await;
                    deserialized_wv.elevator_containers[i].calls.clear();
                }
            }
        }

        // Floor_sensor -> oppdater last_floor_sensor i egen heis-container
        local_network::ElevMsgType::FSENS => {
            print_info(format!("Floor: {:?}", msg.floor_sensor));
            if let (Some(i), Some(floor)) = (self_idx, msg.floor_sensor) {
                deserialized_wv.elevator_containers[i].last_floor_sensor = floor;
            }

        }

        // Stop_button -> funksjon kommer
        local_network::ElevMsgType::SBTN => {
            print_info(format!("Stop button: {:?}", msg.stop_button));

        }

        // Obstruction -> Sett obstruction lik melding fra heis i egen heis-container
        local_network::ElevMsgType::OBSTRX => {
            print_info(format!("Obstruction: {:?}", msg.obstruction));
            if let (Some(i), Some(obs)) = (self_idx, msg.obstruction) {
                deserialized_wv.elevator_containers[i].obstruction = obs;
            }
        }
    }
    *wv = world_view::serialize_worldview(&deserialized_wv);
    true
}

/// ### Oppdaterer egne call-buttons og task_statuses etter de er sent over TCP til master
fn clear_from_sent_tcp(wv: &mut Vec<u8>, tcp_container: Vec<u8>) -> bool {
    let mut deserialized_wv = world_view::deserialize_worldview(&wv);
    let self_idx = world_view::get_index_to_container(utils::SELF_ID.load(Ordering::SeqCst) , wv.clone());
    let tcp_container_des = world_view::deserialize_elev_container(&tcp_container);

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

    if let Some(i) = self_idx {
        /*_____ Fjern Tasks som master har oppdatert _____ */
        deserialized_wv.elevator_containers[i].tasks_status.retain(|t| tasks_ids.contains(&t.id));
        /*_____ Fjern sendte CallButtons _____ */
        deserialized_wv.elevator_containers[i].calls.retain(|call| !tcp_container_des.calls.contains(call));
        *wv = world_view::serialize_worldview(&deserialized_wv);
        return true;
    } else {
        utils::print_cosmic_err("The elevator does not exist clear_sent_container_stuff()".to_string());
        return false;
    }
}

/// ### Gir `task` til slave med `id`
///
/// Ikke ferdig implementert
fn push_task(wv: &mut Vec<u8>, task: Task, id: u8, button: CallButton) -> bool {
    let mut deser_wv = world_view::deserialize_worldview(&wv);

    // Fjern `button` frå `outside_button` om han finst
    if let Some(index) = deser_wv.outside_button.iter().position(|b| *b == button) {
        deser_wv.outside_button.swap_remove(index);
    }

    let self_idx = world_view::get_index_to_container(id, wv.clone());

    if let Some(i) = self_idx {
        // **Hindrar duplikatar: sjekk om task.id allereie finst i `tasks`**
        // NB: skal i teorien være unødvendig å sjekke dette
        if !deser_wv.elevator_containers[i].tasks.iter().any(|t| t.id == task.id) {
            deser_wv.elevator_containers[i].tasks.push(task);
            *wv = world_view::serialize_worldview(&deser_wv);
            return true;
        }
    }

    false
}

/// ### Oppdaterer status til `new_status` til task med `id` i egen heis_container.tasks_status
fn update_task_status(wv: &mut Vec<u8>, task_id: u16, new_status: TaskStatus) -> bool {
    let mut wv_deser = world_view::deserialize_worldview(&wv);
    let self_idx = world_view::get_index_to_container(utils::SELF_ID.load(Ordering::SeqCst), wv.clone());

    if let Some(i) = self_idx {
        // Finner `task` i tasks_status og setter status til `new_status`
        if let Some(task) = wv_deser.elevator_containers[i]
            .tasks_status
            .iter_mut()
            .find(|t| t.id == task_id)
            {
                task.status = new_status.clone();
            }
    }
    // println!("Satt {:?} på id: {}", new_status, task_id);
    *wv = world_view::serialize_worldview(&wv_deser);
    true
}
```

Fil: elevator_pro\src\world_view\world_view_update.rs

```
use crate::network::local_network;
use crate::world_view::world_view;
use crate::network::tcp_network;
use crate::{config, utils};

use std::collections::HashSet;
use std::sync::atomic::{AtomicBool, Ordering};
use std::sync::OnceLock;
use std::thread::sleep;
use std::time::Duration;


static ONLINE: OnceLock<AtomicBool> = OnceLock::new(); // worldview_channel_request
pub fn get_network_status() -> &'static AtomicBool {
    ONLINE.get_or_init(|| AtomicBool::new(false))
}




pub fn join_wv(mut my_wv: Vec<u8>, master_wv: Vec<u8>) -> Vec<u8> {
    //TODO: Lag copy funkjon for worldview structen
    let my_wv_deserialised = world_view::deserialize_worldview(&my_wv);
    let mut master_wv_deserialised = world_view::deserialize_worldview(&master_wv);

    let my_self_index = world_view::get_index_to_container(utils::SELF_ID.load(Ordering::SeqCst) , my_wv);
    let master_self_index = world_view::get_index_to_container(utils::SELF_ID.load(Ordering::SeqCst) , master_wv);


    if let (Some(i_org), Some(i_new)) = (my_self_index, master_self_index) {
        master_wv_deserialised.elevator_containers[i_new].door_open = my_wv_deserialised.elevator_containers[i_org].door_open;
        master_wv_deserialised.elevator_containers[i_new].obstruction = my_wv_deserialised.elevator_containers[i_org].obstruction;
        master_wv_deserialised.elevator_containers[i_new].last_floor_sensor = my_wv_deserialised.elevator_containers[i_org].last_floor_sensor;
        master_wv_deserialised.elevator_containers[i_new].motor_dir = my_wv_deserialised.elevator_containers[i_org].motor_dir;

        master_wv_deserialised.elevator_containers[i_new].calls = my_wv_deserialised.elevator_containers[i_org].calls.clone();

        master_wv_deserialised.elevator_containers[i_new].tasks_status = my_wv_deserialised.elevator_containers[i_org].tasks_status.clone();
        /*Oppdater task_statuses. putt i funksjon hvis det funker?*/
        let new_ids: HashSet<u16> = master_wv_deserialised.elevator_containers[i_new].tasks.iter().map(|t| t.id).collect();
        let old_ids: HashSet<u16> = master_wv_deserialised.elevator_containers[i_new].tasks_status.iter().map(|t|
t.id).collect();

        // Legg til taskar frå masters task som ikkje allereie finst i task_status
        for task in master_wv_deserialised.elevator_containers[i_new].tasks.clone().iter() {
            if !old_ids.contains(&task.id) {
                master_wv_deserialised.elevator_containers[i_new].tasks_status.push(task.clone());
            }
        }
        // Fjern taskar frå task_status som ikkje fins lenger i masters tasks
        master_wv_deserialised.elevator_containers[i_org]
        .tasks_status
        .retain(|t| new_ids.contains(&t.id));


        //Oppdater callbuttons, når master har fått de med seg fjern dine egne
        // Bytter til at vi antar at TCP får frem alle meldinger, og at vi fjerner calls etter vi har sendt på TCP
    } else if let Some(i_org) = my_self_index {
        master_wv_deserialised.add_elev(my_wv_deserialised.elevator_containers[i_org].clone());
    }

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

/// ### Sjekker om vi har internett-tilkobling
pub async fn watch_ethernet() {
    let mut last_net_status = false;
    let mut net_status = false;
    loop {
        let ip = utils::get_self_ip();

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

        if last_net_status != net_status {
            get_network_status().store(net_status, Ordering::SeqCst);
            if net_status {utils::print_ok("Vi er online".to_string());}
            else {utils::print_warn("Vi er offline".to_string());}
            last_net_status = net_status;
        }
    }
}
```