Лекція 21

Actor Model: Реалізація на Tokio

Практична реалізація Actor System

Actor Framework • Typed Actors • Error Handling • Graceful Shutdown

🤖 Повноцінний рій БПЛА на основі Actor Model

Частина 1: Базова інфраструктура

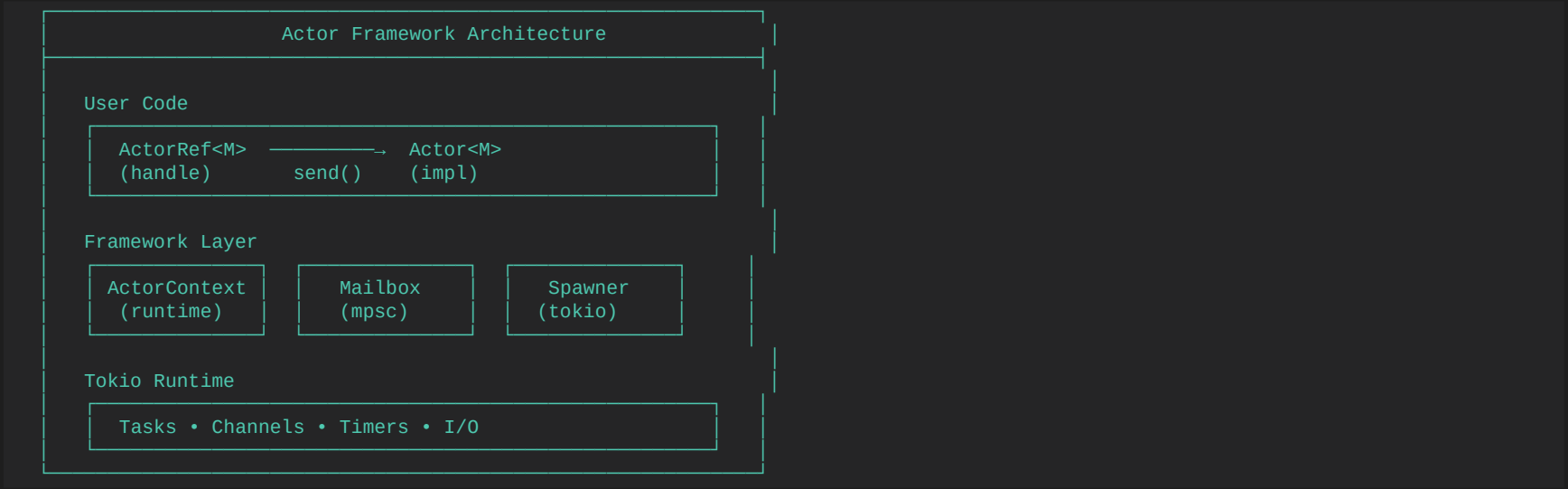
О. С. Бичков • 2025

План лекції (Частина 1)

1. Архітектура Actor Framework
2. ActorContext — контекст виконання
3. ActorRef — типізоване посилання
4. Trait Actor
5. Trait Message
6. Spawning actors
7. Message routing
8. Error types
9. Actor state machine
10. Graceful shutdown
11. Cancellation tokens
12. 🤖 DroneActor trait
13. 🤖 DroneActor implementation
14. 🤖 DroneActorHandle
15. 🤖 Message types
16. Підсумок

Частина 2: Coordinator, Supervision, повна система

Архітектура Actor Framework



```rust
// Основні типи нашого Actor Framework

/// Унікальний ідентифікатор актора
#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash)]
pub struct ActorId(pub u64);

impl ActorId {
    pub fn new() -> Self {
        use std::sync::atomic::{AtomicU64, Ordering};
        static COUNTER: AtomicU64 = AtomicU64::new(0);
        ActorId(COUNTER.fetch_add(1, Ordering::Relaxed))
    }
}

/// Помилки Actor системи
#[derive(Debug, thiserror::Error)]
pub enum ActorError {
    #[error("Mailbox is full")]
    MailboxFull,

    #[error("Actor has stopped")]
    ActorStopped,

    #[error("Request timeout")]
    Timeout,

    #[error("Actor panicked: {0}")]
    Panicked(String),
}
```

```rust
use async_trait::async_trait;

/// Головний trait для всіх акторів
#[async_trait]
pub trait Actor: Send + Sized + 'static {
    /// Тип повідомлень які актор обробляє
    type Message: Send + 'static;

    /// Викликається перед початком обробки повідомлень
    async fn on_start(&mut self, ctx: &mut ActorContext<Self>) {
        // Default: нічого не робити
    }

    /// Обробка повідомлення
    async fn handle(
        &mut self,
        msg: Self::Message,
        ctx: &mut ActorContext<Self>,
    );

    /// Викликається після зупинки актора
    async fn on_stop(&mut self, ctx: &mut ActorContext<Self>) {
        // Default: нічого не робити
    }

    /// Назва актора для логування
    fn name(&self) -> &str {
        std::any::type_name::<Self>()
    }
}
```

```rust
use tokio::sync::mpsc;
use tokio_util::sync::CancellationToken;

/// Контекст в якому виконується актор
pub struct ActorContext<A: Actor> {
    /// ID цього актора
    pub id: ActorId,

    /// Власне посилання (для передачі іншим)
    self_ref: ActorRef<A::Message>,

    /// Токен для graceful shutdown
    cancellation: CancellationToken,

    /// Sender для відправки собі (scheduled messages)
    self_sender: mpsc::Sender<A::Message>,
}

impl<A: Actor> ActorContext<A> {
    /// Отримати посилання на себе
    pub fn self_ref(&self) -> ActorRef<A::Message> {
        self.self_ref.clone()
    }

    /// Запросити зупинку актора
    pub fn stop(&self) {
        self.cancellation.cancel();
    }

    /// Перевірити чи актор має зупинитись
```

```rust
impl<A: Actor> ActorContext<A> {
    /// Створити дочірнього актора
    pub fn spawn<C: Actor>(&self, child: C) -> ActorRef<C::Message> {
        spawn_actor(child)
    }

    /// Запланувати повідомлення з затримкою
    pub fn schedule_once(
        &self,
        delay: Duration,
        msg: A::Message,
    ) {
        let sender = self.self_sender.clone();
        tokio::spawn(async move {
            tokio::time::sleep(delay).await;
            let _ = sender.send(msg).await;
        });
    }

    /// Запланувати періодичні повідомлення
    pub fn schedule_repeat(
        &self,
        interval: Duration,
        msg_fn: impl Fn() -> A::Message + Send + 'static,
    ) {
        let sender = self.self_sender.clone();
        let token = self.cancellation.clone();

        tokio::spawn(async move {
            let mut interval = tokio::time::interval(interval);
```

```rust
use tokio::sync::{mpsc, oneshot};

/// Типізоване посилання на актора
#[derive(Debug)]
pub struct ActorRef<M: Send + 'static> {
    id: ActorId,
    sender: mpsc::Sender<M>,
}

impl<M: Send + 'static> ActorRef<M> {
    /// ID актора
    pub fn id(&self) -> ActorId {
        self.id
    }

    /// Fire-and-forget надсилання
    pub async fn send(&self, msg: M) -> Result<(), ActorError> {
        self.sender.send(msg).await
            .map_err(|_| ActorError::ActorStopped)
    }

    /// Спробувати надіслати без очікування
    pub fn try_send(&self, msg: M) -> Result<(), ActorError> {
        self.sender.try_send(msg).map_err(|e| match e {
            mpsc::error::TrySendError::Full(_) => ActorError::MailboxFull,
            mpsc::error::TrySendError::Closed(_) => ActorError::ActorStopped,
        })
    }

    /// Перевірити чи актор живий
```

```rust
/// Налаштування для spawn
pub struct SpawnConfig {
    pub mailbox_size: usize,
    pub name: Option<String>,
}

impl Default for SpawnConfig {
    fn default() -> Self {
        SpawnConfig {
            mailbox_size: 100,
            name: None,
        }
    }
}

/// Створити та запустити актора
pub fn spawn_actor<A: Actor>(actor: A) -> ActorRef<A::Message> {
    spawn_actor_with_config(actor, SpawnConfig::default())
}

pub fn spawn_actor_with_config<A: Actor>(
    actor: A,
    config: SpawnConfig,
) -> ActorRef<A::Message> {
    let id = ActorId::new();
    let (tx, rx) = mpsc::channel(config.mailbox_size);
    let cancellation = CancellationToken::new();

    let actor_ref = ActorRef { id, sender: tx.clone() };
```

```
/// Головний цикл актора
async fn run_actor<A: Actor>(
    mut actor: A,
    mut receiver: mpsc::Receiver<A::Message>,
    mut ctx: ActorContext<A>,
) {
    // Lifecycle: on_start
    tracing::debug!(actor_id = ?ctx.id, name = actor.name(), "Actor starting");
    actor.on_start(&mut ctx).await;

    // Main message loop
    loop {
        tokio::select! {
            biased;

            // Check cancellation first
            _ = ctx.cancellation.cancelled() => {
                tracing::debug!(actor_id = ?ctx.id, "Actor cancelled");
                break;
            }

            // Process messages
            msg = receiver.recv() => {
                match msg {
                    Some(m) => {
                        actor.handle(m, &mut ctx).await;
                    }
                    None => {
                        // All senders dropped
                        tracing::debug!(actor_id = ?ctx.id, "All senders dropped");
```

```rust
use tokio::sync::oneshot;

/// Trait для повідомлень що очікують відповідь
pub trait Request: Send + 'static {
    type Response: Send + 'static;
}

/// Wrapper для request-reply
pub struct Ask<R: Request> {
    pub request: R,
    pub reply: oneshot::Sender<R::Response>,
}

/// Extension trait для ActorRef
impl<M: Send + 'static> ActorRef<M> {
    /// Надіслати запит і отримати відповідь
    pub async fn ask<R>(
        &self,
        request: R,
        timeout: Duration,
    ) -> Result<R::Response, ActorError>
    where
        R: Request,
        M: From<Ask<R>>,
    {
        let (tx, rx) = oneshot::channel();
        let msg = Ask { request, reply: tx };

        self.send(msg.into()).await?;
```

```rust
// Приклад: повідомлення для Counter Actor

/// Запит на отримання значення
pub struct GetValue;

impl Request for GetValue {
    type Response = i32;
}

/// Всі повідомлення Counter
pub enum CounterMessage {
    Increment,
    Decrement,
    Add(i32),
    GetValue(Ask<GetValue>),
}

// Конвертація для зручності
impl From<Ask<GetValue>> for CounterMessage {
    fn from(ask: Ask<GetValue>) -> Self {
        CounterMessage::GetValue(ask)
    }
}

// Використання:
let counter: ActorRef<CounterMessage> = spawn_actor(Counter::new());

counter.send(CounterMessage::Increment).await?;
counter.send(CounterMessage::Add(10)).await?;
```

```rust
pub struct Counter {
    value: i32,
    name: String,
}

impl Counter {
    pub fn new(name: impl Into<String>) -> Self {
        Counter { value: 0, name: name.into() }
    }
}

#[async_trait]
impl Actor for Counter {
    type Message = CounterMessage;

    async fn on_start(&mut self, ctx: &mut ActorContext<Self>) {
        println!("[{}] Counter started with id {:?}", self.name, ctx.id);
    }

    async fn handle(&mut self, msg: CounterMessage, _ctx: &mut ActorContext<Self>) {
        match msg {
            CounterMessage::Increment => self.value += 1,
            CounterMessage::Decrement => self.value -= 1,
            CounterMessage::Add(n) => self.value += n,
            CounterMessage::GetValue(Ask { request: _, reply }) => {
                let _ = reply.send(self.value);
            }
        }
    }
```

```rust
use tokio_util::sync::CancellationToken;

/// Система для координації shutdown
pub struct ActorSystem {
    shutdown_token: CancellationToken,
    actors: Vec<Box<dyn ActorHandle>>,
}

impl ActorSystem {
    pub fn new() -> Self {
        ActorSystem {
            shutdown_token: CancellationToken::new(),
            actors: Vec::new(),
        }
    }

    /// Запустити shutdown всієї системи
    pub async fn shutdown(&self) {
        tracing::info!("Initiating system shutdown");
        self.shutdown_token.cancel();

        // Чекаємо на завершення всіх акторів
        tokio::time::sleep(Duration::from_secs(5)).await;

        tracing::info!("System shutdown complete");
    }

    /// Чекати на Ctrl+C і shutdown
    pub async fn wait_for_shutdown(&self) {
        tokio::signal::ctrl_c().await.ok();
```

```rust
use tokio::sync::oneshot;

/// Позиція БПЛА
#[derive(Debug, Clone, Copy)]
pub struct Position {
    pub x: f64,
    pub y: f64,
    pub z: f64,
}

/// Статус агента
#[derive(Debug, Clone)]
pub struct DroneStatus {
    pub id: DroneId,
    pub position: Position,
    pub battery: u8,
    pub state: DroneState,
}

/// Команди для БПЛА
#[derive(Debug)]
pub enum DroneMessage {
    // Commands
    MoveTo(Position),
    StartPatrol { area: Area },
    ReturnToBase,
    EmergencyStop,
    Shutdown,

    // Queries
```

```rust
/// Стан БПЛА
#[derive(Debug, Clone)]
pub enum DroneState {
    /// Ініціалізація
    Initializing,

    /// Очікування команд
    Idle,

    /// Рух до точки
    Moving {
        target: Position,
        reason: MoveReason,
    },

    /// Патрулювання
    Patrolling {
        area: Area,
        waypoint_index: usize,
    },

    /// Повернення на базу
    Returning {
        base: Position,
    },

    /// Аварійна зупинка
    Emergency {
        reason: String,
    },
```

```rust
pub struct DroneActor {
    // Ідентифікація
    id: DroneId,
    name: String,

    // Стан
    position: Position,
    state: DroneState,
    battery: u8,

    // Конфігурація
    config: DroneConfig,
    base_position: Position,

    // Комунікація
    coordinator: Option<ActorRef<CoordinatorMessage>>,
}

#[derive(Debug, Clone)]
pub struct DroneConfig {
    pub speed: f64,
    pub scan_radius: f64,
    pub low_battery_threshold: u8,
    pub critical_battery_threshold: u8,
    pub tick_interval: Duration,
}

impl Default for DroneConfig {
    fn default() -> Self {
        DroneConfig {
```

```rust
impl DroneActor {
    pub fn new(
        id: DroneId,
        config: DroneConfig,
        base_position: Position,
    ) -> Self {
        DroneActor {
            id,
            name: format!("Drone-{}", id.0),
            position: base_position,
            state: DroneState::Initializing,
            battery: 100,
            config,
            base_position,
            coordinator: None,
        }
    }

    pub fn with_coordinator(mut self, coord: ActorRef<CoordinatorMessage>) -> Self {
        self.coordinator = Some(coord);
        self
    }
}

#[async_trait]
impl Actor for DroneActor {
    type Message = DroneMessage;

    async fn on_start(&mut self, ctx: &mut ActorContext<Self>) {
        tracing::info!(drone_id = ?self.id, "Drone starting");
```

```rust
#[async_trait]
impl Actor for DroneActor {
    // ... on_start ...

    async fn handle(&mut self, msg: DroneMessage, ctx: &mut ActorContext<Self>) {
        match msg {
            DroneMessage::MoveTo(target) => {
                self.state = DroneState::Moving {
                    target,
                    reason: MoveReason::Command,
                };
            }

            DroneMessage::StartPatrol { area } => {
                self.state = DroneState::Patrolling {
                    area,
                    waypoint_index: 0,
                };
            }

            DroneMessage::ReturnToBase => {
                self.state = DroneState::Returning { base: self.base_position };
            }

            DroneMessage::EmergencyStop => {
                self.state = DroneState::Emergency {
                    reason: "Emergency stop commanded".to_string(),
                };
            }
```

```rust
impl DroneActor {
    async fn on_tick(&mut self, ctx: &mut ActorContext<Self>) {
        // Оновлення батареї
        self.battery = self.battery.saturating_sub(1);

        // Перевірка критичного рівня батареї
        if self.battery <= self.config.critical_battery_threshold {
            self.state = DroneState::Returning { base: self.base_position };
        }

        // Логіка в залежності від стану
        match &self.state {
            DroneState::Moving { target, .. } => {
                self.move_towards(*target);
                if self.reached(*target) {
                    self.state = DroneState::Idle;
                }
            }

            DroneState::Patrolling { area, waypoint_index } => {
                let waypoint = area.waypoint(*waypoint_index);
                self.move_towards(waypoint);

                if self.reached(waypoint) {
                    self.state = DroneState::Patrolling {
                        area: area.clone(),
                        waypoint_index: (waypoint_index + 1) % area.waypoint_count(),
                    };
                }
            }
```

```rust
impl DroneActor {
    fn move_towards(&mut self, target: Position) {
        let dx = target.x - self.position.x;
        let dy = target.y - self.position.y;
        let dz = target.z - self.position.z;
        let distance = (dx*dx + dy*dy + dz*dz).sqrt();

        if distance > self.config.speed {
            let ratio = self.config.speed / distance;
            self.position.x += dx * ratio;
            self.position.y += dy * ratio;
            self.position.z += dz * ratio;
        } else {
            self.position = target;
        }
    }

    fn reached(&self, target: Position) -> bool {
        let dx = target.x - self.position.x;
        let dy = target.y - self.position.y;
        let dz = target.z - self.position.z;
        (dx*dx + dy*dy + dz*dz).sqrt() < 1.0
    }

    fn get_status(&self) -> DroneStatus {
        DroneStatus {
            id: self.id,
            position: self.position,
            battery: self.battery,
            state: self.state.clone(),
```

🕹️ MAS DroneHandle — зручний API

```rust
/// Зручний handle для роботи з DroneActor
#[derive(Clone)]
pub struct DroneHandle {
    id: DroneId,
    actor_ref: ActorRef<DroneMessage>,
}

impl DroneHandle {
    pub fn spawn(
        id: DroneId,
        config: DroneConfig,
        base: Position,
        coordinator: Option<ActorRef<CoordinatorMessage>>,
    ) -> Self {
        let mut actor = DroneActor::new(id, config, base);
        if let Some(coord) = coordinator {
            actor = actor.with_coordinator(coord);
        }

        let actor_ref = spawn_actor(actor);
        DroneHandle { id, actor_ref }
    }

    pub fn id(&self) -> DroneId { self.id }

    pub async fn move_to(&self, pos: Position) -> Result<(), ActorError> {
        self.actor_ref.send(DroneMessage::MoveTo(pos)).await
    }

    pub async fn get_status(&self) -> Result<DroneStatus, ActorError> {
```

Підсумок: Частина 1

Actor Framework компоненти:
- Actor trait — інтерфейс актора
- ActorRef<M> — типізоване посилання
- ActorContext — контекст виконання
- spawn_actor() — створення актора

Patterns:
- Request-Reply з Ask<R>
- Scheduled messages (once, repeat)
- Graceful shutdown з CancellationToken

🤖 DroneActor:
- State machine (DroneState)
- Tick-based update loop
- Report to Coordinator
- Typed handle API

→ Частина 2: Coordinator, Supervision, повна система

Лекція 21 (продовження)

Actor Model: Coordinator та System

Координатор, Supervision, повна система рою

Coordinator • Registry • Supervisor • EventBus • SwarmSystem

🤖 Production-ready рій БПЛА

Частина 2: Coordinator та System

О. С. Бичков • 2025

План лекції (Частина 2)

1. CoordinatorMessage
2. CoordinatorActor
3. Coordinator handle
4. Drone Registry
5. Event Bus Actor
6. SwarmEvent types
7. Supervisor Actor
8. Supervision strategies
9. SwarmSystem struct
10. System initialization
11. 🤖 Full drone swarm
12. 🤖 Mission assignment
13. 🤖 Target detection flow
14. 🤖 Testing the system
15. Best practices
16. Підсумок

```rust
/// Повідомлення для Coordinator
#[derive(Debug)]
pub enum CoordinatorMessage {
    // === Управління дронами ===
    RegisterDrone {
        drone_id: DroneId,
        handle: DroneHandle,
        reply: oneshot::Sender<Result<(), CoordinatorError>>,
    },
    UnregisterDrone {
        drone_id: DroneId,
    },

    // === Звіти від дронів ===
    DroneReport {
        drone_id: DroneId,
        status: DroneStatus,
    },
    TargetDetected {
        drone_id: DroneId,
        target: Target,
    },

    // === Місії ===
    AssignMission {
        mission: Mission,
        reply: oneshot::Sender<Result<DroneId, CoordinatorError>>,
    },

    // === Запити ===
```

```rust
use std::collections::HashMap;

pub struct CoordinatorActor {
    // Реєстр дронів
    drones: HashMap<DroneId, DroneEntry>,

    // Активні місії
    missions: HashMap<MissionId, MissionEntry>,

    // Event Bus для broadcast
    event_bus: Option<ActorRef<EventBusMessage>>,

    // Статистика
    stats: CoordinatorStats,
}

struct DroneEntry {
    handle: DroneHandle,
    last_status: Option<DroneStatus>,
    last_update: Instant,
    assigned_mission: Option<MissionId>,
}

struct MissionEntry {
    mission: Mission,
    assigned_drone: Option<DroneId>,
    status: MissionStatus,
}

#[derive(Default)]
```

```rust
#[async_trait]
impl Actor for CoordinatorActor {
    type Message = CoordinatorMessage;

    async fn on_start(&mut self, ctx: &mut ActorContext<Self>) {
        tracing::info!("Coordinator starting");

        // Periodic health check
        ctx.schedule_repeat(Duration::from_secs(5), || {
            CoordinatorMessage::HealthCheck
        });
    }

    async fn handle(&mut self, msg: CoordinatorMessage, ctx: &mut ActorContext<Self>) {
        match msg {
            CoordinatorMessage::RegisterDrone { drone_id, handle, reply } => {
                let result = self.register_drone(drone_id, handle);
                let _ = reply.send(result);
            }

            CoordinatorMessage::DroneReport { drone_id, status } => {
                self.update_drone_status(drone_id, status);
            }

            CoordinatorMessage::TargetDetected { drone_id, target } => {
                self.handle_target_detected(drone_id, target, ctx).await;
            }

            CoordinatorMessage::AssignMission { mission, reply } => {
                let result = self.assign_mission(mission).await;
```

```
impl CoordinatorActor {
    fn register_drone(&mut self, id: DroneId, handle: DroneHandle) -> Result<(), CoordinatorError> {
        if self.drones.contains_key(&id) {
            return Err(CoordinatorError::DroneAlreadyRegistered(id));
        }

        self.drones.insert(id, DroneEntry {
            handle,
            last_status: None,
            last_update: Instant::now(),
            assigned_mission: None,
        });

        self.stats.total_drones += 1;
        tracing::info!(drone_id = ?id, "Drone registered");
        Ok(())
    }

    fn update_drone_status(&mut self, id: DroneId, status: DroneStatus) {
        if let Some(entry) = self.drones.get_mut(&id) {
            entry.last_status = Some(status);
            entry.last_update = Instant::now();
        }
    }

    fn find_available_drone(&self, target_pos: Position) -> Option<DroneId> {
        self.drones.iter()
            .filter(|(_, entry)| {
                entry.assigned_mission.is_none() &&
                entry.last_status.as_ref()
```

```rust
impl CoordinatorActor {
    async fn assign_mission(&mut self, mission: Mission) -> Result<DroneId, CoordinatorError> {
        // Знаходимо найкращого дрона
        let drone_id = self.find_available_drone(mission.target_area.center())
            .ok_or(CoordinatorError::NoDroneAvailable)?;

        // Призначаємо місію
        let mission_id = mission.id;
        self.missions.insert(mission_id, MissionEntry {
            mission: mission.clone(),
            assigned_drone: Some(drone_id),
            status: MissionStatus::Assigned,
        });

        // Оновлюємо дрона
        if let Some(entry) = self.drones.get_mut(&drone_id) {
            entry.assigned_mission = Some(mission_id);

            // Надсилаємо команду дрону
            entry.handle.start_patrol(mission.target_area.clone()).await?;
        }

        self.stats.active_missions += 1;

        tracing::info!(
            mission_id = ?mission_id,
            drone_id = ?drone_id,
            "Mission assigned"
        );
```

🎯 MAS Target Detection Flow

```
impl CoordinatorActor {
    async fn handle_target_detected(
        &mut self,
        drone_id: DroneId,
        target: Target,
        ctx: &ActorContext<Self>,
    ) {
        self.stats.targets_detected += 1;

        tracing::warn!(
            drone_id = ?drone_id,
            target_id = ?target.id,
            position = ?target.position,
            "Target detected!"
        );

        // Broadcast event
        if let Some(event_bus) = &self.event_bus {
            let _ = event_bus.send(EventBusMessage::Publish(
                SwarmEvent::TargetDetected {
                    detector: drone_id,
                    target: target.clone(),
                }
            )).await;
        }

        // Призначаємо додаткових дронів для підтримки
        if target.threat_level >= ThreatLevel::High {
            self.request_support(drone_id, target.position).await;
        }
```

MAC Coordinator Handle

```rust
#[derive(Clone)]
pub struct CoordinatorHandle {
    actor_ref: ActorRef<CoordinatorMessage>,
}

impl CoordinatorHandle {
    pub fn spawn(event_bus: Option<ActorRef<EventBusMessage>>) -> Self {
        let actor = CoordinatorActor::new(event_bus);
        let actor_ref = spawn_actor(actor);
        CoordinatorHandle { actor_ref }
    }

    pub async fn register_drone(
        &self,
        drone_id: DroneId,
        handle: DroneHandle,
    ) -> Result<(), CoordinatorError> {
        let (tx, rx) = oneshot::channel();
        self.actor_ref.send(CoordinatorMessage::RegisterDrone {
            drone_id, handle, reply: tx,
        }).await?;
        rx.await.map_err(|_| CoordinatorError::ActorStopped)?
    }

    pub async fn assign_mission(&self, mission: Mission) -> Result<DroneId, CoordinatorError> {
        let (tx, rx) = oneshot::channel();
        self.actor_ref.send(CoordinatorMessage::AssignMission {
            mission, reply: tx,
        }).await?;
        rx.await.map_err(|_| CoordinatorError::ActorStopped)?
```

```rust
use tokio::sync::broadcast;

#[derive(Debug, Clone)]
pub enum SwarmEvent {
    DroneRegistered(DroneId),
    DroneUnregistered(DroneId),
    TargetDetected { detector: DroneId, target: Target },
    MissionAssigned { mission_id: MissionId, drone_id: DroneId },
    MissionCompleted { mission_id: MissionId },
    AlertLevel(AlertLevel),
    SystemShutdown,
}

pub enum EventBusMessage {
    Publish(SwarmEvent),
    Subscribe(oneshot::Sender<broadcast::Receiver<SwarmEvent>>),
}

pub struct EventBusActor {
    sender: broadcast::Sender<SwarmEvent>,
}

#[async_trait]
impl Actor for EventBusActor {
    type Message = EventBusMessage;

    async fn handle(&mut self, msg: EventBusMessage, _ctx: &mut ActorContext<Self>) {
        match msg {
            EventBusMessage::Publish(event) => {
                let _ = self.sender.send(event);
```

```rust
use std::any::Any;
use std::collections::HashMap;

pub enum RegistryMessage {
    Register {
        name: String,
        actor: Box<dyn Any + Send + Sync>,
        reply: oneshot::Sender<Result<(), RegistryError>>,
    },
    Lookup {
        name: String,
        reply: oneshot::Sender<Option<Box<dyn Any + Send + Sync>>>,
    },
    Unregister {
        name: String,
    },
}

pub struct RegistryActor {
    actors: HashMap<String, Box<dyn Any + Send + Sync>>,
}

impl RegistryActor {
    pub fn new() -> Self {
        RegistryActor { actors: HashMap::new() }
    }
}

#[async_trait]
impl Actor for RegistryActor {
```

```rust
use tokio::task::JoinSet;

pub struct SupervisorActor {
    children: JoinSet<SupervisedResult>,
    child_specs: HashMap<ActorId, ChildSpec>,
    strategy: SupervisionStrategy,
    max_restarts: u32,
    restart_window: Duration,
    restart_counts: HashMap<ActorId, Vec<Instant>>,
}

#[derive(Clone)]
pub enum SupervisionStrategy {
    OneForOne,
    OneForAll,
}

struct ChildSpec {
    factory: Box<dyn Fn() -> BoxFuture<'static, SupervisedResult> + Send + Sync>,
}

impl SupervisorActor {
    async fn supervise(&mut self) {
        loop {
            if let Some(result) = self.children.join_next().await {
                match result {
                    Ok(Ok(actor_id)) => {
                        tracing::info!(actor_id = ?actor_id, "Child completed normally");
                    }
                    Ok(Err((actor_id, error))) => {
```

```
impl SupervisorActor {
    async fn handle_failure(&mut self, actor_id: ActorId, error: ActorError) {
        tracing::warn!(actor_id = ?actor_id, error = ?error, "Child failed");

        // Check restart limits
        let restarts = self.restart_counts.entry(actor_id).or_default();
        let now = Instant::now();

        // Remove old restarts outside window
        restarts.retain(|t| now.duration_since(*t) < self.restart_window);

        if restarts.len() >= self.max_restarts as usize {
            tracing::error!(actor_id = ?actor_id, "Max restarts exceeded, stopping");
            self.child_specs.remove(&actor_id);
            return;
        }

        // Apply strategy
        match self.strategy {
            SupervisionStrategy::OneForOne => {
                self.restart_child(actor_id).await;
            }
            SupervisionStrategy::OneForAll => {
                self.restart_all_children().await;
            }
        }

        restarts.push(now);
    }
```

```rust
/// Повна система рою БПЛА
pub struct SwarmSystem {
    // Core actors
    coordinator: CoordinatorHandle,
    event_bus: ActorRef<EventBusMessage>,
    registry: ActorRef<RegistryMessage>,

    // Drones
    drones: HashMap<DroneId, DroneHandle>,

    // Shutdown coordination
    shutdown_token: CancellationToken,
}

impl SwarmSystem {
    pub async fn new() -> Self {
        // Create event bus first
        let event_bus = spawn_actor(EventBusActor::new());

        // Create registry
        let registry = spawn_actor(RegistryActor::new());

        // Create coordinator with event bus
        let coordinator = CoordinatorHandle::spawn(Some(event_bus.clone()));

        SwarmSystem {
            coordinator,
            event_bus,
            registry,
            drones: HashMap::new(),
```

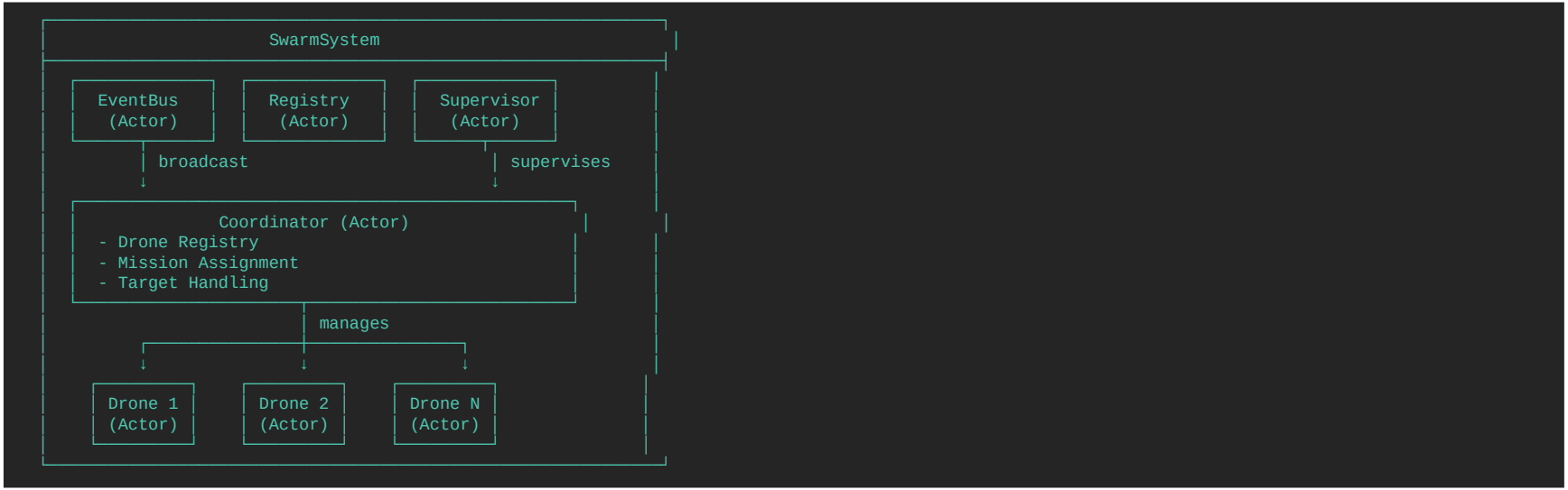
```rust
impl SwarmSystem {
    /// Додати нового дрона до рою
    pub async fn spawn_drone(
        &mut self,
        id: DroneId,
        config: DroneConfig,
        base: Position,
    ) -> Result<DroneHandle, SwarmError> {
        // Create drone actor
        let drone = DroneHandle::spawn(
            id,
            config,
            base,
            Some(self.coordinator.actor_ref.clone()),
        );

        // Register with coordinator
        self.coordinator.register_drone(id, drone.clone()).await?;

        // Store locally
        self.drones.insert(id, drone.clone());

        tracing::info!(drone_id = ?id, "Drone spawned and registered");

        Ok(drone)
    }

    /// Spawn кількох дронів
    pub async fn spawn_drone_fleet(
        &mut self,
```

```rust
impl SwarmSystem {
    /// Subscribe to swarm events
    pub async fn subscribe_events(&self) -> broadcast::Receiver<SwarmEvent> {
        let (tx, rx) = oneshot::channel();
        self.event_bus.send(EventBusMessage::Subscribe(tx)).await.ok();
        rx.await.unwrap()
    }

    /// Assign a mission to the swarm
    pub async fn assign_mission(&self, mission: Mission) -> Result<DroneId, SwarmError> {
        self.coordinator.assign_mission(mission).await
            .map_err(SwarmError::Coordinator)
    }

    /// Graceful shutdown
    pub async fn shutdown(&mut self) {
        tracing::info!("Initiating swarm shutdown");

        // Broadcast shutdown event
        let _ = self.event_bus.send(EventBusMessage::Publish(
            SwarmEvent::SystemShutdown
        )).await;

        // Shutdown coordinator (it will shutdown all drones)
        let _ = self.coordinator.shutdown().await;

        // Wait for cleanup
        tokio::time::sleep(Duration::from_secs(2)).await;

        tracing::info!("Swarm shutdown complete");
```

```rust
#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Initialize logging
    tracing_subscriber::fmt::init();

    tracing::info!("Starting Drone Swarm System");

    // Create swarm system
    let mut swarm = SwarmSystem::new().await;

    // Subscribe to events
    let mut events = swarm.subscribe_events().await;
    tokio::spawn(async move {
        while let Ok(event) = events.recv().await {
            tracing::info!("Event: {:?}", event);
        }
    });

    // Spawn drone fleet
    let base_positions = (0..10).map(|i| Position {
        x: (i % 5) as f64 * 100.0,
        y: (i / 5) as f64 * 100.0,
        z: 0.0,
    });

    swarm.spawn_drone_fleet(10, DroneConfig::default(), base_positions).await?;

    // Assign a patrol mission
    let mission = Mission::new_patrol(Area::new(
        Position::new(0.0, 0.0, 50.0),
```

```rust
#[cfg(test)]
mod tests {
    use super::*;

    #[tokio::test]
    async fn test_drone_actor() {
        let drone = DroneHandle::spawn(
            DroneId(1),
            DroneConfig::default(),
            Position::new(0.0, 0.0, 0.0),
            None,
        );

        // Test get_status
        let status = drone.get_status().await.unwrap();
        assert!(matches!(status.state, DroneState::Idle));
        assert_eq!(status.battery, 100);

        // Test move_to
        drone.move_to(Position::new(100.0, 0.0, 0.0)).await.unwrap();
        tokio::time::sleep(Duration::from_millis(500)).await;

        let status = drone.get_status().await.unwrap();
        assert!(matches!(status.state, DroneState::Moving { .. }));

        // Test shutdown
        drone.shutdown().await.unwrap();
    }

    #[tokio::test]
```

MAC: Повна архітектура системи



🤖 MAC: Message Flow — Target Detection

Потік при виявленні цілі:

1. Drone tick() → scan_environment()
2. Drone виявляє Target → send to Coordinator
   DroneMessage::TargetDetected { drone_id, target }

3. Coordinator обробляє:
   - Оновлює статистику
   - Broadcast через EventBus
   - Якщо high threat → request_support()

4. EventBus broadcast:
   SwarmEvent::TargetDetected { detector, target }

5. Інші компоненти отримують подію:
   - Dashboard оновлюється
   - Logger записує
   - AlertSystem перевіряє threat level

6. Coordinator призначає support drones:
   - find_available_drones()
   - send MoveTo commands

Best Practices

✅ Actor Design:
- Single responsibility per actor
- Immutable messages
- Handle pattern для API
- Timeout на всі request-reply

✅ System Design:
- EventBus для loose coupling
- Registry для discovery
- Supervisor для fault tolerance
- Graceful shutdown

❌ Avoid:
- Blocking in handle()
- Circular dependencies
- Large messages (use Arc)
- Unbounded mailboxes in production

🤖 MAC Best Practices:
- Periodic health checks
- Battery monitoring
- Mission timeout handling

Performance Considerations

Mailbox Sizing:
- Commands: 100-500
- Telemetry: 1000-5000
- Events: 100-1000

Message Throughput:
- Small messages: ~1M/sec
- With async handling: ~100K/sec
- With I/O: depends on I/O

Actor Count:
- Thousands of actors: OK
- Memory: ~KB per actor
- Tokio handles scheduling

Optimizations:
- Batch messages where possible
- Use try_send for non-critical
- Monitor mailbox length
- Profile with tokio-console

Підсумок лекції

Actor Framework на Tokio:
- Actor trait + ActorRef + ActorContext
- Request-Reply з Ask pattern
- Scheduled messages (once, repeat)

System Components:
- Coordinator — центральне управління
- EventBus — broadcast events
- Registry — actor discovery
- Supervisor — fault tolerance

🤖 SwarmSystem:
- Full drone fleet management
- Mission assignment
- Target detection flow
- Graceful shutdown

→ Наступна лекція: Rayon — Data Parallelism

Завдання для самостійної роботи

1. Counter Actor (базове):
   - Increment, Decrement, GetValue
   - Tests

2. Timer Actor:
   - Schedule once/repeat
   - Cancel scheduled

3. Drone Actor:
   - State machine
   - Tick-based movement
   - Battery simulation

4. Coordinator:
   - Drone registry
   - Mission assignment
   - Status queries

5. Full Swarm:
   - 10 дронів
   - EventBus
   - Target detection
   - Graceful shutdown
   - Integration tests

Actor System опановано!

Coordinator • EventBus • Supervisor • SwarmSystem

Питання?