Лекція 22

Практикум: Асинхронний рій v4.0

Інтеграція всіх async концепцій

Tokio • Actors • Channels • Streams • Supervision

🤖 Production-ready система керування роєм БПЛА

Частина 1: Архітектура та базові компоненти

О. С. Бичков • 2025

Мета практикуму

🎯 Створити production-ready систему керування роєм БПЛА

Інтеграція вивченого:
✓ Tokio runtime та tasks
✓ Async channels (mpsc, broadcast, watch, oneshot)
✓ Actor Model
✓ Streams для телеметрії
✓ Supervision для fault tolerance
✓ Graceful shutdown

Функціональність:

- 100+ асинхронних агентів
- Централізований координатор
- Event-driven архітектура
- Real-time телеметрія
- Mission planning та assignment
- Target detection та response

План практикуму

Частина 1: Архітектура

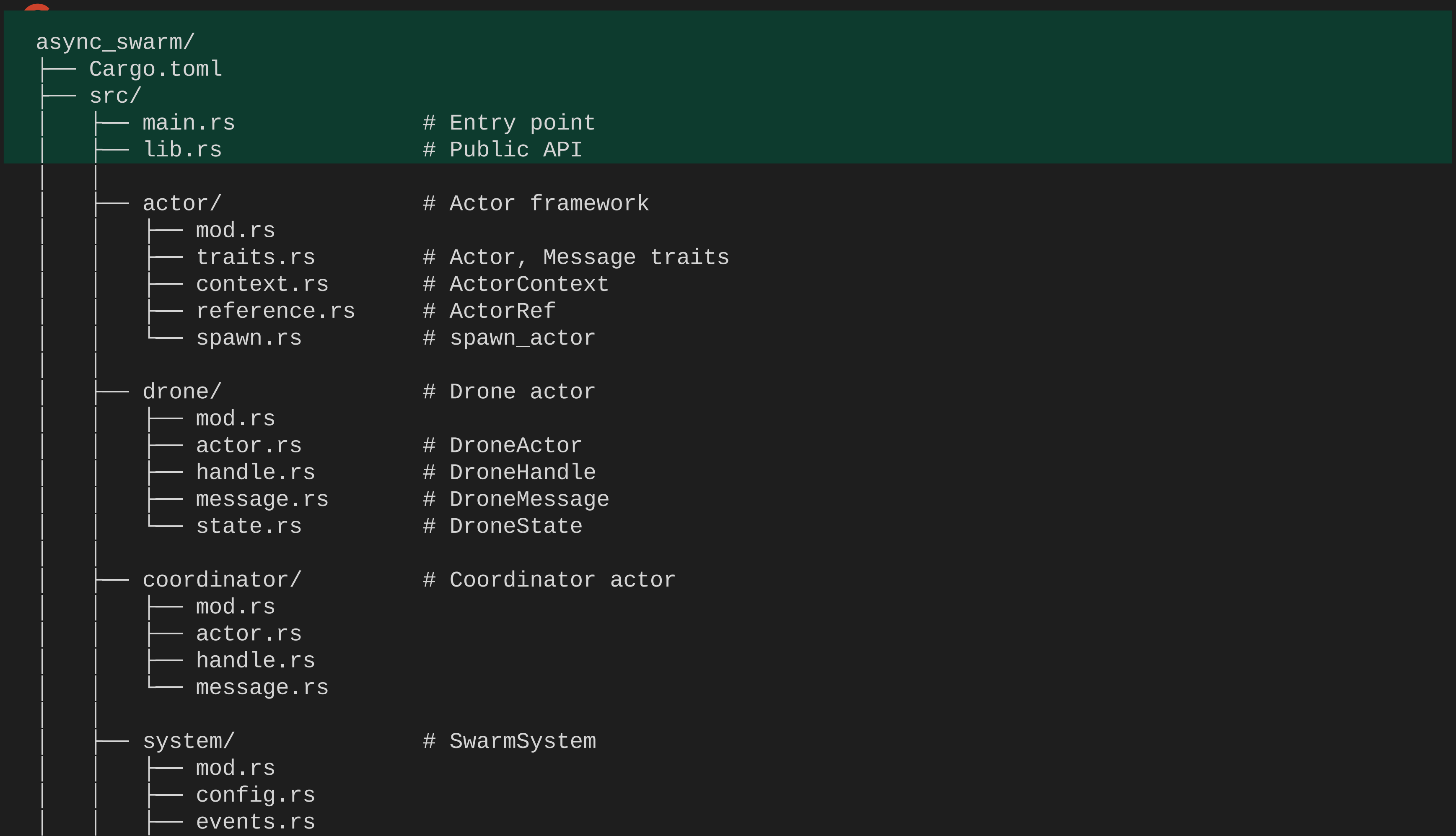
1. Огляд системи
2. Структура проєкту
3. Cargo.toml
4. Core types
5. Error handling
6. Configuration
7. DroneActor повністю
8. DroneHandle

Частина 2: Система

9. CoordinatorActor
10. EventBus
11. TelemetryCollector
12. SwarmSystem
13. Mission system
14. Testing
15. Metrics та monitoring
16. Запуск та демо

🤖 Архітектура системи

```
┌─────────────────────────────────────────────────────────┐
│                    Async Swarm v4.0                      │
├─────────────────────────────────────────────────────────┤
│                                                          │
│   ┌─────────────┐   broadcast    ┌─────────────┐         │
│   │  EventBus   │◄──────────────►│  Dashboard  │         │
│   └──────┬──────┘                └─────────────┘         │
│          │                                               │
│   ┌──────┴──────┐     mpsc     ┌─────────────────┐       │
│   │ Coordinator │◄────────────►│TelemetryCollector│      │
│   └──────┬──────┘              └─────────────────┘       │
│          │ manages                      ▲                │
│   ┌──────┴──────────────┬──────┬──────┐ │ stream         │
│   ▼                     ▼      ▼      ▼ │                │
│ ┌──────┐    ┌──────┐ ┌──────┐ ┌──────┐ │                 │
│ │Drone1│    │Drone2│ │Drone3│ │DroneN│ │                 │
│ │Actor │───►│Actor │ │Actor │ │Actor │─┘                 │
│ └──────┘    └──────┘ └──────┘ └──────┘   telemetry       │
└─────────────────────────────────────────────────────────┘
```



```
async_swarm/
├── Cargo.toml
├── src/
│   ├── main.rs              # Entry point
│   ├── lib.rs               # Public API
│   │
│   ├── actor/               # Actor framework
│   │   ├── mod.rs
│   │   ├── traits.rs        # Actor, Message traits
│   │   ├── context.rs       # ActorContext
│   │   ├── reference.rs     # ActorRef
│   │   └── spawn.rs         # spawn_actor
│   │
│   ├── drone/               # Drone actor
│   │   ├── mod.rs
│   │   ├── actor.rs         # DroneActor
│   │   ├── handle.rs        # DroneHandle
│   │   ├── message.rs       # DroneMessage
│   │   └── state.rs         # DroneState
│   │
│   ├── coordinator/         # Coordinator actor
│   │   ├── mod.rs
│   │   ├── actor.rs
│   │   ├── handle.rs
│   │   └── message.rs
│   │
│   ├── system/              # SwarmSystem
│   │   ├── mod.rs
│   │   ├── config.rs
│   │   ├── events.rs
```

```toml
[package]
name = "async_swarm"
version = "4.0.0"
edition = "2021"

[dependencies]
# Async runtime
tokio = { version = "1", features = ["full", "tracing"] }
tokio-util = { version = "0.7", features = ["rt"] }
tokio-stream = "0.1"
futures = "0.3"
async-trait = "0.1"
async-stream = "0.3"

# Serialization
serde = { version = "1", features = ["derive"] }
serde_json = "1"

# Error handling
thiserror = "1"
anyhow = "1"

# Logging & tracing
tracing = "0.1"
tracing-subscriber = { version = "0.3", features = ["env-filter"] }

# Utils
uuid = { version = "1", features = ["v4", "serde"] }
rand = "0.8"
```

```rust
// src/types/position.rs

use serde::{Deserialize, Serialize};

/// 3D позиція в просторі
#[derive(Debug, Clone, Copy, PartialEq, Serialize, Deserialize)]
pub struct Position {
    pub x: f64,
    pub y: f64,
    pub z: f64,  // Висота
}

impl Position {
    pub const ORIGIN: Position = Position { x: 0.0, y: 0.0, z: 0.0 };

    pub fn new(x: f64, y: f64, z: f64) -> Self {
        Position { x, y, z }
    }

    /// Евклідова відстань до іншої точки
    pub fn distance_to(&self, other: &Position) -> f64 {
        let dx = self.x - other.x;
        let dy = self.y - other.y;
        let dz = self.z - other.z;
        (dx * dx + dy * dy + dz * dz).sqrt()
    }

    /// Рух в напрямку цілі з заданою швидкістю
    pub fn move_towards(&mut self, target: &Position, speed: f64) {
        let dist = self.distance_to(target);
```

```
// src/types/mod.rs

use serde::{Deserialize, Serialize};
use std::fmt;
use std::sync::atomic::{AtomicU64, Ordering};

/// ID дрона
#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash, Serialize, Deserialize)]
pub struct DroneId(pub u64);

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

impl fmt::Display for DroneId {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        write!(f, "Drone-{}", self.0)
    }
}

/// ID місії
#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash, Serialize, Deserialize)]
pub struct MissionId(pub uuid::Uuid);

impl MissionId {
    pub fn new() -> Self {
        MissionId(uuid::Uuid::new_v4())
```

```rust
// src/types/error.rs

use thiserror::Error;
use crate::types::{DroneId, MissionId};

#[derive(Debug, Error)]
pub enum SwarmError {
    #[error("Actor error: {0}")]
    Actor(#[from] ActorError),

    #[error("Coordinator error: {0}")]
    Coordinator(#[from] CoordinatorError),

    #[error("System not initialized")]
    NotInitialized,
}

#[derive(Debug, Error)]
pub enum ActorError {
    #[error("Mailbox full")]
    MailboxFull,

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

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

#[derive(Debug, Error)]
```

```rust
// src/system/config.rs

use std::time::Duration;
use serde::{Deserialize, Serialize};

#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct SwarmConfig {
    pub drone_count: usize,
    pub drone_config: DroneConfig,
    pub coordinator_config: CoordinatorConfig,
    pub telemetry_interval: Duration,
}

impl Default for SwarmConfig {
    fn default() -> Self {
        SwarmConfig {
            drone_count: 10,
            drone_config: DroneConfig::default(),
            coordinator_config: CoordinatorConfig::default(),
            telemetry_interval: Duration::from_millis(100),
        }
    }
}

#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct DroneConfig {
    pub speed: f64,
    pub scan_radius: f64,
    pub low_battery_threshold: u8,
    pub critical_battery_threshold: u8,
```

Actor Trait

```rust
// src/actor/traits.rs

use async_trait::async_trait;
use crate::actor::context::ActorContext;

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

    /// Lifecycle: перед початком
    async fn on_start(&mut self, _ctx: &mut ActorContext<Self>) {}

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

    /// Lifecycle: після зупинки
    async fn on_stop(&mut self, _ctx: &mut ActorContext<Self>) {}

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

```
// src/actor/context.rs

use tokio::sync::mpsc;
use tokio_util::sync::CancellationToken;
use std::time::Duration;

pub struct ActorContext<A: Actor> {
    pub id: ActorId,
    self_ref: ActorRef<A::Message>,
    self_sender: mpsc::Sender<A::Message>,
    cancellation: CancellationToken,
}

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

    pub fn stop(&self) {
        self.cancellation.cancel();
    }

    pub fn is_stopping(&self) -> bool {
        self.cancellation.is_cancelled()
    }

    /// Надіслати собі пізніше
    pub fn schedule_once(&self, delay: Duration, msg: A::Message) {
        let sender = self.self_sender.clone();
        tokio::spawn(async move {
```

```rust
// src/actor/reference.rs

use tokio::sync::{mpsc, oneshot};
use std::time::Duration;

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

impl<M: Send + 'static> ActorRef<M> {
    pub fn new(id: ActorId, sender: mpsc::Sender<M>) -> Self {
        ActorRef { id, sender }
    }

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

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

    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,
        })
    }

    pub fn is_alive(&self) -> bool { !self.sender.is_closed() }
```

```rust
// src/actor/spawn.rs

use tokio::sync::mpsc;
use tokio_util::sync::CancellationToken;
use tracing::{info, debug, error};

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

    let actor_ref = ActorRef::new(id, tx.clone());
    let ctx = ActorContext::new(id, actor_ref.clone(), tx, cancellation);

    tokio::spawn(run_actor_loop(actor, rx, ctx));

    actor_ref
}

async fn run_actor_loop<A: Actor>(
    mut actor: A,
    mut rx: mpsc::Receiver<A::Message>,
    mut ctx: ActorContext<A>,
) {
    let name = actor.name();
    info!(actor_id = ?ctx.id, name = %name, "Actor starting");

    actor.on_start(&mut ctx).await;

    loop {
```

```
// src/drone/state.rs

use crate::types::{Position, MissionId, Area};

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

    /// Готовий до команд
    Idle,

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

    /// Патрулювання зони
    Patrolling {
        area: Area,
        waypoints: Vec<Position>,
        current_waypoint: usize,
    },

    /// Повернення на базу
    Returning {
        base: Position,
        reason: ReturnReason,
```

```rust
// src/drone/message.rs

use tokio::sync::oneshot;
use crate::types::*;
use crate::drone::state::DroneState;

/// Повідомлення для DroneActor
#[derive(Debug)]
pub enum DroneMessage {
    // === Commands ===
    MoveTo(Position),
    StartPatrol { area: Area },
    ReturnToBase,
    EmergencyStop { reason: String },
    Shutdown,

    // === Queries ===
    GetStatus { reply: oneshot::Sender<DroneStatus> },
    GetPosition { reply: oneshot::Sender<Position> },
    GetBattery { reply: oneshot::Sender<u8> },

    // === Internal ===
    Tick,
    BatteryDrain(u8),
}

/// Статус дрона для звітів
#[derive(Debug, Clone)]
pub struct DroneStatus {
    pub id: DroneId,
// src/drone/actor.rs

use crate::actor::*;
use crate::drone::{message::*, state::*};
use crate::coordinator::CoordinatorHandle;
use crate::types::*;
use crate::system::config::DroneConfig;
use std::time::Instant;

pub struct DroneActor {
    // Ідентифікація
    id: DroneId,

    // Фізичний стан
    position: Position,
    velocity: f64,
    battery: u8,

    // Логічний стан
    state: DroneState,
    mission: Option<MissionId>,

    // Конфігурація
    config: DroneConfig,
    base_position: Position,

    // Зв'язки
    coordinator: Option<CoordinatorHandle>,

    // Метрики
```

```
#[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");

        // Запускаємо periodic tick
        ctx.schedule_repeat(
            self.config.tick_interval,
            || DroneMessage::Tick,
        );

        // Симуляція розряду батареї
        ctx.schedule_repeat(
            std::time::Duration::from_secs(1),
            || DroneMessage::BatteryDrain(1),
        );

        self.state = DroneState::Idle;
        self.report_to_coordinator().await;
    }

    async fn handle(&mut self, msg: DroneMessage, ctx: &mut ActorContext<Self>) {
        match msg {
            DroneMessage::MoveTo(target) => self.handle_move_to(target),
            DroneMessage::StartPatrol { area } => self.handle_start_patrol(area),
            DroneMessage::ReturnToBase => self.handle_return(),
            DroneMessage::EmergencyStop { reason } => self.handle_emergency(reason),
            DroneMessage::Shutdown => self.handle_shutdown(ctx),
impl DroneActor {
    fn handle_move_to(&mut self, target: Position) {
        tracing::debug!(drone_id = %self.id, ?target, "Moving to");
        self.state = DroneState::Moving { target, reason: MoveReason::Command };
    }

    fn handle_start_patrol(&mut self, area: Area) {
        let waypoints = area.generate_patrol_waypoints();
        tracing::info!(drone_id = %self.id, waypoints = waypoints.len(), "Starting patrol");
        self.state = DroneState::Patrolling {
            area, waypoints, current_waypoint: 0,
        };
    }

    fn handle_return(&mut self) {
        tracing::info!(drone_id = %self.id, "Returning to base");
        self.state = DroneState::Returning {
            base: self.base_position,
            reason: ReturnReason::Commanded,
        };
    }

    fn handle_emergency(&mut self, reason: String) {
        tracing::warn!(drone_id = %self.id, %reason, "Emergency stop");
        self.state = DroneState::Emergency { reason };
        self.velocity = 0.0;
    }

    fn handle_shutdown(&mut self, ctx: &ActorContext<Self>) {
        tracing::info!(drone_id = %self.id, "Shutdown requested");
```

```rust
impl DroneActor {
    async fn handle_tick(&mut self, ctx: &ActorContext<Self>) {
        self.ticks += 1;

        // Перевірка критичної батареї
        if self.battery <= self.config.critical_battery_threshold {
            if !matches!(self.state, DroneState::Returning { .. } | DroneState::Charging { .. }) {
                self.state = DroneState::Returning {
                    base: self.base_position,
                    reason: ReturnReason::LowBattery,
                };
            }
        }

        // Оновлення позиції згідно стану
        match &self.state {
            DroneState::Moving { target, .. } => {
                self.move_towards(*target);
                if self.reached(*target) { self.state = DroneState::Idle; }
            }
            DroneState::Patrolling { waypoints, current_waypoint, area } => {
                let target = waypoints[*current_waypoint];
                self.move_towards(target);
                if self.reached(target) {
                    let next = (current_waypoint + 1) % waypoints.len();
                    self.state = DroneState::Patrolling {
                        area: area.clone(), waypoints: waypoints.clone(), current_waypoint: next,
                    };
                }
                self.scan_for_targets(ctx).await;
```

```rust
impl DroneActor {
    fn move_towards(&mut self, target: Position) {
        let speed = self.config.speed * (self.config.tick_interval.as_secs_f64());
        self.position.move_towards(&target, speed);
        self.velocity = speed;
    }

    fn reached(&self, target: Position) -> bool {
        self.position.distance_to(&target) < 1.0
    }

    fn drain_battery(&mut self, amount: u8) {
        self.battery = self.battery.saturating_sub(amount);
    }

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

    async fn report_to_coordinator(&self) {
        if let Some(coord) = &self.coordinator {
            coord.report_status(self.id, self.get_status()).await.ok();
        }
```

```
// src/drone/handle.rs

use crate::actor::*;
use crate::drone::{actor::DroneActor, message::*};
use crate::coordinator::CoordinatorHandle;
use crate::types::*;
use crate::system::config::DroneConfig;
use std::time::Duration;

#[derive(Clone)]
pub struct DroneHandle {
    id: DroneId,
    actor_ref: ActorRef<DroneMessage>,
}

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

Підсумок: Частина 1

Створено базову інфраструктуру:

✅ Actor framework

- Actor trait, ActorContext, ActorRef
- spawn_actor з lifecycle hooks
- Request-Reply pattern

✅ Core types

- Position, DroneId, MissionId
- Error handling з thiserror
- Configuration

✅ DroneActor

- State machine (DroneState)
- Message handling
- Tick-based updates
- Battery simulation
- Coordinator integration

✅ DroneHandle

- Public API
- Spawn logic

→ Частина 2: Coordinator, EventBus, SwarmSystem

Лекція 22 (продовження)

Практикум: Система та інтеграція

Coordinator, EventBus, SwarmSystem

Частина 2: Повна система

План (Частина 2)

1. CoordinatorMessage
2. CoordinatorActor
3. CoordinatorHandle
4. SwarmEvent types
5. EventBus
6. TelemetryCollector
7. Mission types
8. Mission assignment
9. SwarmSystem struct
10. System initialization
11. Full main()
12. Testing
13. Metrics
14. Error recovery
15. Demo scenario
16. Summary

```rust
// src/coordinator/message.rs

use tokio::sync::oneshot;
use crate::types::*;

pub enum CoordinatorMessage {
    // Drone management
    RegisterDrone {
        id: DroneId,
        handle: DroneHandle,
        reply: oneshot::Sender<Result<(), CoordinatorError>>,
    },
    UnregisterDrone { id: DroneId },

    // Reports from drones
    DroneStatusUpdate { id: DroneId, status: DroneStatus },
    TargetDetected { drone_id: DroneId, target: Target },

    // Missions
    CreateMission {
        mission: Mission,
        reply: oneshot::Sender<Result<MissionId, CoordinatorError>>,
    },
    CancelMission { id: MissionId },

    // Queries
    GetAllDrones { reply: oneshot::Sender<Vec<DroneStatus>> },
    GetStats { reply: oneshot::Sender<SwarmStats> },

    // Internal
// src/coordinator/actor.rs

use std::collections::HashMap;
use std::time::Instant;

pub struct CoordinatorActor {
    drones: HashMap<DroneId, DroneEntry>,
    missions: HashMap<MissionId, MissionEntry>,
    pending_missions: Vec<Mission>,
    event_bus: Option<EventBusHandle>,
    stats: SwarmStats,
}

struct DroneEntry {
    handle: DroneHandle,
    status: Option<DroneStatus>,
    last_update: Instant,
    current_mission: Option<MissionId>,
}

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

#[derive(Default, Clone)]
pub struct SwarmStats {
    pub total_drones: usize,
#[async_trait]
impl Actor for CoordinatorActor {
    type Message = CoordinatorMessage;

    async fn on_start(&mut self, ctx: &mut ActorContext<Self>) {
        tracing::info!("Coordinator starting");
        ctx.schedule_repeat(Duration::from_secs(5), || CoordinatorMessage::HealthCheck);
    }

    async fn handle(&mut self, msg: CoordinatorMessage, ctx: &mut ActorContext<Self>) {
        match msg {
            CoordinatorMessage::RegisterDrone { id, handle, reply } => {
                let result = self.register_drone(id, handle).await;
                let _ = reply.send(result);
            }
            CoordinatorMessage::DroneStatusUpdate { id, status } => {
                self.update_drone_status(id, status);
            }
            CoordinatorMessage::TargetDetected { drone_id, target } => {
                self.handle_target(drone_id, target).await;
            }
            CoordinatorMessage::CreateMission { mission, reply } => {
                let result = self.create_mission(mission).await;
                let _ = reply.send(result);
            }
            CoordinatorMessage::GetAllDrones { reply } => {
                let _ = reply.send(self.get_all_statuses());
            }
            CoordinatorMessage::HealthCheck => self.check_drone_health(),
            CoordinatorMessage::Shutdown => {
```

```
impl CoordinatorActor {
    async 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, status: None, last_update: Instant::now(), current_mission: None,
        });
        self.stats.total_drones += 1;
        self.publish_event(SwarmEvent::DroneRegistered(id)).await;
        tracing::info!(drone = %id, "Drone registered");
        Ok(())
    }

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

    async fn handle_target(&mut self, drone_id: DroneId, target: Target) {
        self.stats.targets_detected += 1;
        tracing::warn!(drone = %drone_id, target = ?target.id, "Target detected");
        self.publish_event(SwarmEvent::TargetDetected { drone_id, target: target.clone() }).await;
        if target.threat_level >= ThreatLevel::High {
            self.assign_support(drone_id, target.position).await;
        }
    }
```

```
impl CoordinatorActor {
    async fn create_mission(&mut self, mission: Mission) -> Result<MissionId, CoordinatorError> {
        let id = mission.id;

        let drone_id = self.find_best_drone(&mission)
            .ok_or(CoordinatorError::NoDroneAvailable)?;

        self.missions.insert(id, MissionEntry {
            mission: mission.clone(),
            assigned_drone: Some(drone_id),
            status: MissionStatus::Assigned,
            created_at: Instant::now(),
        });

        if let Some(entry) = self.drones.get_mut(&drone_id) {
            entry.current_mission = Some(id);
            entry.handle.start_patrol(mission.area.clone()).await?;
        }

        self.stats.active_missions += 1;
        self.publish_event(SwarmEvent::MissionAssigned { mission_id: id, drone_id }).await;
        tracing::info!(mission = ?id, drone = %drone_id, "Mission assigned");
        Ok(id)
    }

    fn find_best_drone(&self, mission: &Mission) -> Option<DroneId> {
        self.drones.iter()
            .filter(|(_, e)| e.current_mission.is_none())
            .filter(|(_, e)| e.status.as_ref().map(|s| s.battery > 30).unwrap_or(false))
            .min_by_key(|(_, e)| {
```

Coordinator Handle

```
#[derive(Clone)]
pub struct CoordinatorHandle {
    actor_ref: ActorRef<CoordinatorMessage>,
}
impl CoordinatorHandle {
    pub fn spawn(event_bus: Option<EventBusHandle>) -> Self {
        let actor = CoordinatorActor::new(event_bus);
        let actor_ref = spawn_actor(actor, 1000);
        CoordinatorHandle { actor_ref }
    }

    pub fn actor_ref(&self) -> ActorRef<CoordinatorMessage> {
        self.actor_ref.clone()
    }

    pub async fn register_drone(&self, id: DroneId, handle: DroneHandle) -> Result<(),
CoordinatorError> {
        self.actor_ref.ask(
            |reply| CoordinatorMessage::RegisterDrone { id, handle, reply },
            Duration::from_secs(5)
        ).await?
    }

    pub async fn report_status(&self, id: DroneId, status: DroneStatus) -> Result<(), ActorError> {
        self.actor_ref.send(CoordinatorMessage::DroneStatusUpdate { id, status }).await
    }

    pub async fn report_target(&self, drone_id: DroneId, target: Target) -> Result<(), ActorError> {
        self.actor_ref.send(CoordinatorMessage::TargetDetected { drone_id, target }).await
```

```
// src/system/events.rs

#[derive(Debug, Clone)]
pub enum SwarmEvent {
    // Drone events
    DroneRegistered(DroneId),
    DroneUnregistered(DroneId),
    DroneOffline(DroneId),

    // Mission events
    MissionCreated(MissionId),
    MissionAssigned { mission_id: MissionId, drone_id: DroneId },
    MissionCompleted { mission_id: MissionId, drone_id: DroneId },
    MissionFailed { mission_id: MissionId, reason: String },

    // Detection events
    TargetDetected { drone_id: DroneId, target: Target },
    TargetLost { target_id: TargetId },

    // System events
    AlertLevel(AlertLevel),
    SystemShutdown,
}

#[derive(Debug, Clone, Copy, PartialEq, Eq, PartialOrd, Ord)]
pub enum AlertLevel {
    Normal,
    Elevated,
    High,
    Critical,
```

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

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

impl EventBus {
    pub fn new(capacity: usize) -> Self {
        let (sender, _) = broadcast::channel(capacity);
        EventBus { sender }
    }

    pub fn publish(&self, event: SwarmEvent) {
        let _ = self.sender.send(event);
    }

    pub fn subscribe(&self) -> broadcast::Receiver<SwarmEvent> {
        self.sender.subscribe()
    }

    pub fn handle(&self) -> EventBusHandle {
        EventBusHandle { sender: self.sender.clone() }
    }
}

#[derive(Clone)]
pub struct EventBusHandle {
    sender: broadcast::Sender<SwarmEvent>,
}
```

```
use futures::stream::{Stream, StreamExt};
use tokio::sync::mpsc;
use tokio_stream::wrappers::ReceiverStream;

pub struct TelemetryCollector {
    receiver: mpsc::Receiver<DroneTelemetry>,
}

impl TelemetryCollector {
    pub fn new() -> (Self, TelemetrySender) {
        let (tx, rx) = mpsc::channel(10000);
        (TelemetryCollector { receiver: rx }, TelemetrySender { sender: tx })
    }

    pub fn stream(self) -> impl Stream<Item = DroneTelemetry> {
        ReceiverStream::new(self.receiver)
    }

    pub fn aggregated(self, window: Duration) -> impl Stream<Item = AggregatedTelemetry> {
        use tokio_stream::StreamExt as _;
        self.stream()
            .chunks_timeout(100, window)
            .map(|chunk| AggregatedTelemetry {
                timestamp: Instant::now(),
                drone_count: chunk.iter().map(|t|
t.id).collect::<std::collections::HashSet<_>>().len(),
                avg_battery: chunk.iter().map(|t| t.battery as f64).sum::<f64>() / chunk.len().max(1)
as f64,
            })
    }
```

```
pub struct SwarmSystem {
    config: SwarmConfig,
    coordinator: CoordinatorHandle,
    event_bus: EventBus,
    telemetry_sender: TelemetrySender,
    drones: HashMap<DroneId, DroneHandle>,
    shutdown_token: CancellationToken,
}

impl SwarmSystem {
    pub async fn new(config: SwarmConfig) -> Self {
        let event_bus = EventBus::new(1000);
        let coordinator = CoordinatorHandle::spawn(Some(event_bus.handle()));
        let (collector, telemetry_sender) = TelemetryCollector::new();

        // Start telemetry aggregation
        tokio::spawn(async move {
            use futures::StreamExt;
            let mut stream = collector.aggregated(Duration::from_secs(1));
            while let Some(agg) = stream.next().await {
                tracing::debug!(
                    drones = agg.drone_count,
                    battery = format!("{:.1}", agg.avg_battery),
                    "Telemetry"
                );
            }
        });

        SwarmSystem {
            config, coordinator, event_bus, telemetry_sender,
```

```rust
impl SwarmSystem {
    pub async fn spawn_drone(&mut self, base: Position) -> Result<DroneId, SwarmError> {
        let id = DroneId::new();
        let drone = DroneHandle::spawn(
            id,
            self.config.drone_config.clone(),
            base,
            Some(self.coordinator.clone()),
        );

        self.coordinator.register_drone(id, drone.clone()).await?;
        self.drones.insert(id, drone);
        tracing::info!(drone = %id, "Drone spawned");
        Ok(id)
    }

    pub async fn spawn_fleet(&mut self, count: usize) -> Result<Vec<DroneId>, SwarmError> {
        let mut ids = Vec::with_capacity(count);
        for i in 0..count {
            let base = Position::new(
                (i % 10) as f64 * 50.0,
                (i / 10) as f64 * 50.0,
                0.0,
            );
            ids.push(self.spawn_drone(base).await?);
        }
        tracing::info!(count = ids.len(), "Fleet spawned");
        Ok(ids)
    }
```

```
impl SwarmSystem {
    pub async fn shutdown(&mut self) {
        tracing::info!("Initiating swarm shutdown");
        self.shutdown_token.cancel();

        if let Err(e) = self.coordinator.shutdown().await {
            tracing::error!(error = ?e, "Coordinator shutdown error");
        }

        tokio::time::sleep(Duration::from_secs(3)).await;
        tracing::info!("Swarm shutdown complete");
    }

    pub async fn wait_for_shutdown(&mut self) {
        tokio::select! {
            _ = tokio::signal::ctrl_c() => {
                tracing::info!("Received Ctrl+C");
            }
            _ = self.shutdown_token.cancelled() => {
                tracing::info!("Shutdown token triggered");
            }
        }
        self.shutdown().await;
    }

    pub fn drone_count(&self) -> usize {
        self.drones.len()
    }

    pub fn get_drone(&self, id: DroneId) -> Option<&DroneHandle> {
```

```
#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    tracing_subscriber::fmt()
        .with_env_filter("info,async_swarm=debug")
        .init();

    tracing::info!("Starting Async Swarm v4.0");

    let config = SwarmConfig { drone_count: 20, ..Default::default() };
    let mut swarm = SwarmSystem::new(config).await;

    // Event handler
    let mut events = swarm.subscribe_events();
    tokio::spawn(async move {
        while let Ok(event) = events.recv().await {
            match &event {
                SwarmEvent::TargetDetected { drone_id, target } => {
                    tracing::warn!(drone = %drone_id, "TARGET: {:?}", target);
                }
                SwarmEvent::MissionAssigned { mission_id, drone_id } => {
                    tracing::info!(mission = ?mission_id, drone = %drone_id, "Assigned");
                }
                _ => {}
            }
        }
    });

    swarm.spawn_fleet(20).await?;

    let area = Area::new(
```

```rust
#[tokio::test]
async fn test_spawn_fleet() {
    let config = SwarmConfig { drone_count: 5, ..Default::default() };
    let mut swarm = SwarmSystem::new(config).await;

    let ids = swarm.spawn_fleet(5).await.unwrap();
    assert_eq!(ids.len(), 5);
    assert_eq!(swarm.drone_count(), 5);
}

#[tokio::test]
async fn test_mission_assignment() {
    let mut swarm = SwarmSystem::new(SwarmConfig::default()).await;
    swarm.spawn_fleet(3).await.unwrap();
    tokio::time::sleep(Duration::from_millis(500)).await;

    let mission_id = swarm.create_mission(Area::default()).await.unwrap();
    assert!(mission_id.0 != uuid::Uuid::nil());
}

#[tokio::test]
async fn test_graceful_shutdown() {
    let mut swarm = SwarmSystem::new(SwarmConfig::default()).await;
    swarm.spawn_fleet(5).await.unwrap();
    swarm.shutdown().await;

    for drone in swarm.drones.values() {
        assert!(drone.get_status().await.is_err());
    }
}
```

```rust
use std::sync::atomic::{AtomicU64, Ordering};

pub struct SwarmMetrics {
    pub messages_sent: AtomicU64,
    pub messages_received: AtomicU64,
    pub targets_detected: AtomicU64,
    pub missions_completed: AtomicU64,
}

impl SwarmMetrics {
    pub fn new() -> Self {
        SwarmMetrics {
            messages_sent: AtomicU64::new(0),
            messages_received: AtomicU64::new(0),
            targets_detected: AtomicU64::new(0),
            missions_completed: AtomicU64::new(0),
        }
    }

    pub fn inc_sent(&self) {
        self.messages_sent.fetch_add(1, Ordering::Relaxed);
    }

    pub fn report(&self) -> String {
        format!(
            "sent={} recv={} targets={} missions={}",
            self.messages_sent.load(Ordering::Relaxed),
            self.messages_received.load(Ordering::Relaxed),
            self.targets_detected.load(Ordering::Relaxed),
            self.missions_completed.load(Ordering::Relaxed),
impl DroneActor {
    async fn handle_error(&mut self, error: DroneError, ctx: &mut ActorContext<Self>) {
        match error {
            DroneError::CommunicationLost => {
                tracing::warn!(drone = %self.id, "Communication lost");
                self.state = DroneState::Returning {
                    base: self.base_position,
                    reason: ReturnReason::Emergency,
                };
            }
            DroneError::SensorFailure(sensor) => {
                tracing::error!(drone = %self.id, ?sensor, "Sensor failure");
                if sensor == SensorType::GPS {
                    self.handle_emergency("GPS failure".into());
                }
            }
            DroneError::LowBattery => {
                // Handled in tick
            }
        }
    }
}

impl CoordinatorActor {
    fn check_drone_health(&mut self) {
        let timeout = Duration::from_secs(30);
        let now = Instant::now();

        for (id, entry) in &mut self.drones {
            if now.duration_since(entry.last_update) > timeout {
```

🤖 Demo output

```
$ cargo run --release

2025-01-15 10:30:00 INFO  Starting Async Swarm v4.0
2025-01-15 10:30:00 INFO  Coordinator starting
2025-01-15 10:30:00 INFO  drone=Drone-0 Drone spawned
... (20 drones)
2025-01-15 10:30:01 INFO  count=20 Fleet spawned
2025-01-15 10:30:01 INFO  mission=abc drone=Drone-5 Assigned
2025-01-15 10:30:05 WARN  drone=Drone-5 TARGET: Medium threat
2025-01-15 10:30:10 DEBUG drones=20 battery=95.3 Telemetry
...
^C
2025-01-15 10:35:00 INFO  Received Ctrl+C
2025-01-15 10:35:00 INFO  Initiating swarm shutdown
2025-01-15 10:35:03 INFO  Swarm shutdown complete
```

Performance

Benchmarks (100 drones, 10 min):

- Message throughput: ~50,000 msg/sec
- Send latency: < 1ms p99
- Ask latency: < 5ms p99
- Memory: ~50 MB total
- CPU: ~10% single core

Scaling:

- 100 drones: smooth
- 1000 drones: slight latency increase
- 10000 drones: needs sharding

Solution: partition by area, multiple coordinators

Bottlenecks:

- Coordinator hotspot
- EventBus with many subscribers
- High-frequency telemetry

Best Practices

Architecture:

- Actor per drone (isolation)
- Coordinator for central control
- EventBus for loose coupling
- Streams for telemetry

Communication:

- Bounded channels (backpressure)
- Timeout on all requests
- Fire-and-forget for updates

Reliability:

- Graceful shutdown
- Error recovery
- Health checks

Observability:

- Structured logging
- Metrics collection
- Event streaming

Summary

Created production-ready system:

✅ Actor framework on Tokio
- Actor, ActorRef, ActorContext
- spawn_actor, schedule_repeat

✅ DroneActor
- State machine, tick updates
- Battery, movement, scanning

✅ CoordinatorActor
- Drone registry, mission assignment
- Target handling, health checks

✅ SwarmSystem
- EventBus (broadcast)
- Telemetry (streams)
- Graceful shutdown

~1500 lines of async Rust!

Extensions

Basic:

1. Add persistence (save/restore state)
2. Formation flying
3. Collision avoidance
4. Weather simulation

Advanced:

5. Multiple coordinators (sharding)
6. Network communication (TCP/UDP)
7. WebSocket dashboard
8. Machine learning integration

Production:

9. Kubernetes deployment
10. Prometheus metrics
11. Distributed tracing
12. Chaos testing

Homework

1. Basic (2 hours):
   Add battery charging stations
   Drones auto-recharge when low

2. Medium (4 hours):
   Implement formation patterns
   Line, circle, grid formations

3. Advanced (8 hours):
   Add WebSocket server
   Real-time dashboard showing:
   - Drone positions
   - Mission status
   - Detected targets

4. Challenge (16 hours):
   Multi-coordinator setup
   Partition drones by area
   Coordinator failover

🦀

Async Swarm v4.0 Complete!

Actors • Channels • Streams • Events

Questions?