Лекція 20

# Actor Model: Концепції

## Модель акторів для конкурентних систем

Actor • Message • Mailbox • Supervision • Location Transparency

🤖 Агент = Актор: природна відповідність для MAC

Частина 1: Теоретичні основи

О. С. Бичков • 2025

# План лекції (Частина 1)

1. Історія та мотивація
2. Що таке Actor?
3. Три аксіоми Actor Model
4. Повідомлення та Mailbox
5. Asynchronous message passing
6. Actor lifecycle
7. Порівняння з OOP
8. Порівняння з Threads
9. Порівняння з CSP
10. Переваги Actor Model
11. Недоліки та обмеження
12. 🤖 Agent як Actor
13. 🤖 Swarm як Actor System
14. Actor в екосистемі Rust
15. Коли використовувати
16. Підсумок

Частина 2: Реалізація на Tokio, патерни, supervision

# Історія Actor Model

1973 — Carl Hewitt, MIT
Перша публікація "A Universal Modular ACTOR Formalism"

1986 — Gul Agha
"Actors: A Model of Concurrent Computation"
Формалізація та теоретичні основи

1995 — Erlang
Перша промислова мова з Actor Model
Ericsson — телекомунікаційні системи
"Nine 9s" reliability (99.9999999% uptime)

2009 — Akka (Scala/Java)
Popularization in JVM world

"Everything is an Actor" — концептуально схоже до "Everything is an Object"

2010s — Elixir, Pony, Orleans
Різні реалізації та адаптації

# Чому Actor Model?

Проблеми традиційного підходу (shared state + locks):

❌ Race conditions — важко виявити та відтворити
❌ Deadlocks — взаємне блокування
❌ Priority inversion — низькопріоритетний блокує високо
❌ Складність reasoning — важко міркувати про стан
❌ Погана масштабованість — contention на locks

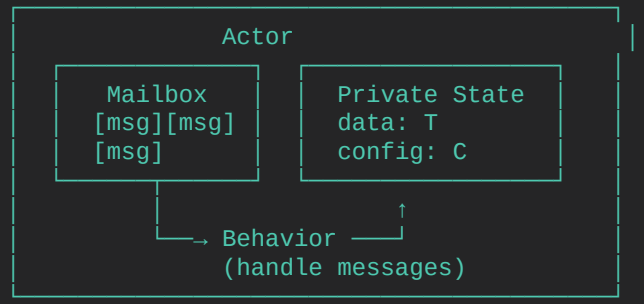
Рішення Actor Model:

✓ No shared state — кожен актор має власний стан
✓ Message passing — єдиний спосіб комунікації
✓ Sequential processing — один message за раз
✓ Location transparency — не важливо де актор
✓ Supervision — ієрархія для обробки помилок
✓ Легке масштабування — актори незалежні

# Що таке Actor?

Actor — фундаментальна одиниця обчислень в Actor Model

Кожен Actor має:




- Mailbox — черга вхідних повідомлень
- Private State — ізольований внутрішній стан
- Behavior — логіка обробки повідомлень

```rust
// Приклад трьох аксіом
impl Actor for MyActor {
    fn handle(&mut self, msg: Message, ctx: &mut Context) {
        // 1. SEND — надіслати повідомлення
        ctx.send(other_actor, Response::Ok);

        // 2. CREATE — створити нового актора
        let child = ctx.spawn(ChildActor::new());

        // 3. DESIGNATE — змінити поведінку/стан
        self.state = NewState::Processing;
    }
}
```

```
// Типізовані повідомлення як enum
enum AgentMessage {
    // Команди (fire-and-forget)
    MoveTo(Position),
    StartPatrol(Area),
    Stop,

    // Запити (очікують відповідь)
    GetStatus { reply_to: ActorRef<StatusResponse> },
    GetPosition { reply_to: ActorRef<Position> },

    // Події (notification)
    TargetDetected(Target),
    BatteryLow(u8),
}

// Відповіді
enum StatusResponse {
    Ok(AgentStatus),
    Error(String),
}
```

# Mailbox — черга повідомлень

Mailbox — буфер для вхідних повідомлень актора

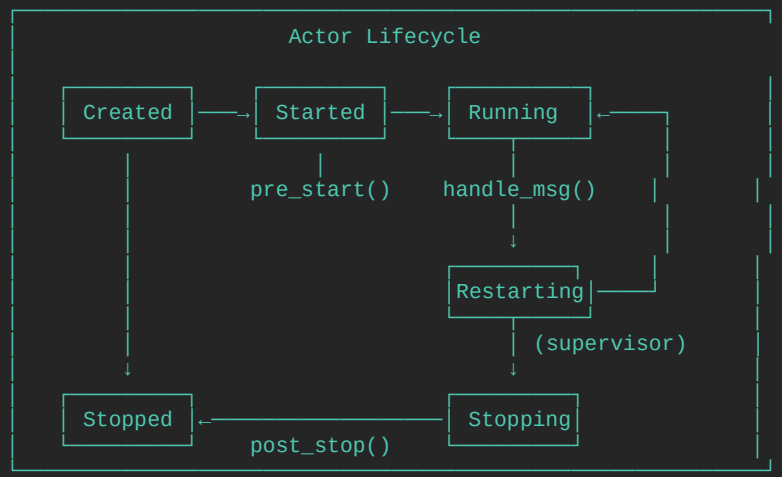
Типи Mailbox:
- Unbounded — необмежена черга (ризик OOM)
- Bounded — з обмеженням (backpressure)
- Priority — з пріоритетами повідомлень

```
Обробка повідомлень:

  Sender A ──msg1──┐
  Sender B ──msg2──┼──→ [Mailbox: msg1, msg2, msg3] ──→ Actor
  Sender C ──msg3──┘                                    (one at a time)

• Повідомлення обробляються ПОСЛІДОВНО
• Один message за раз — no race conditions!
• Actor ніколи не переривається під час обробки
```

```rust
// АСИНХРОННЕ надсилання — sender НЕ чекає
async fn fire_and_forget() {
    // Надсилаємо і продовжуємо
    actor.send(Message::DoWork).await;  // Тільки кладе в mailbox

    // Не чекаємо на обробку!
    println!("Message sent, continuing...");
}

// Request-Reply pattern (якщо потрібна відповідь)
async fn request_reply() {
    // Створюємо канал для відповіді
    let (tx, rx) = oneshot::channel();

    // Надсилаємо з reply channel
    actor.send(Message::GetStatus { reply_to: tx }).await;

    // Чекаємо на відповідь
    let status = rx.await?;
}

// Tell vs Ask:
// Tell: actor.tell(msg)   — fire-and-forget
// Ask:  actor.ask(msg)    — чекає відповідь
```

Важливо: send() не гарантує доставку чи обробку!

# Actor Lifecycle — життєвий цикл



```rust
trait Actor {
    type Message;

    /// Викликається перед початком обробки повідомлень
    async fn pre_start(&mut self, ctx: &mut Context<Self>) {
        // Ініціалізація ресурсів
        // Підписка на події
        // Запуск таймерів
    }

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

    /// Викликається після зупинки
    async fn post_stop(&mut self, ctx: &mut Context<Self>) {
        // Очищення ресурсів
        // Закриття з'єднань
        // Збереження стану
    }

    /// Викликається перед перезапуском (після помилки)
    async fn pre_restart(&mut self, error: &Error, ctx: &mut Context<Self>) {
        // Логування помилки
        // Очищення перед рестартом
    }
}
```

# Actor Model vs OOP

| Аспект | OOP | Actor Model |
|---|---|---|
| Одиниця | Object | Actor |
| Комунікація | Method call (sync) | Message (async) |
| Стан | Може бути shared | Завжди private |
| Concurrency | Locks, mutexes | Built-in (mailbox) |
| Помилки | Exceptions bubble up | Supervision hierarchy |
| Ідентичність | Reference | Address/ActorRef |
| Створення | new() | spawn() |

OOP: object.method(args) — синхронний виклик
Actor: actor.send(message) — асинхронне повідомлення

Actor Model можна розглядати як "OOP done right" для concurrency

# Actor Model vs Threads

| Аспект | Threads + Mutex | Actors |
|---|---|---|
| State protection | Manual (locks) | Automatic (isolation) |
| Communication | Shared memory | Messages |
| Scheduling | OS preemptive | Runtime cooperative |
| Memory | Heavy (~2MB/thread) | Light (~KB/actor) |
| Scalability | ~10K threads | ~1M+ actors |
| Deadlocks | Possible | Impossible* |
| Debugging | Hard (race cond.) | Easier (deterministic) |

* Deadlock between actors неможливий через async messages, але можливий logical deadlock

Actor = lightweight "virtual thread" з ізольованим станом

# Actor Model vs CSP (Go channels)

| Аспект | Actor Model | CSP (Go) |
|---|---|---|
| First-class entity | Actor | Channel |
| Identity | Actor has address | Channel is anonymous |
| State | Inside actor | Inside goroutine |
| Communication | Actor-to-actor | Through channel |
| Mailbox | Per actor | Shared channel |
| Supervision | Built-in hierarchy | Manual |
| Failure handling | Let it crash | Error handling |

Actor: "Надішли повідомлення актору A"
CSP: "Надішли значення в канал C"

Go: "Don't communicate by sharing memory; share memory by communicating"
Actor: "Communicate by sending messages to named actors"

# Переваги Actor Model

- ✓ Ізоляція стану — no shared state, no race conditions
- ✓ Простота reasoning — actor = state machine
- ✓ Масштабованість — легко додавати акторів
- ✓ Location transparency — actor може бути локально або remote
- ✓ Fault tolerance — supervision trees для recovery
- ✓ Природня модель для distributed systems
- ✓ Легше тестування — детерміновані unit tests
- ✓ Hot code upgrade — можна оновлювати поведінку

🤖 Ідеально для MAC — Agent ≈ Actor!

# Недоліки та обмеження

❌ Overhead — message passing дорожчий за method call

❌ Debugging — важко трейсити потік повідомлень

❌ No shared state — іноді потрібен для performance

❌ Message ordering — гарантується тільки між парою

❌ Boilerplate — багато коду для messages/handlers

❌ Learning curve — інша парадигма мислення

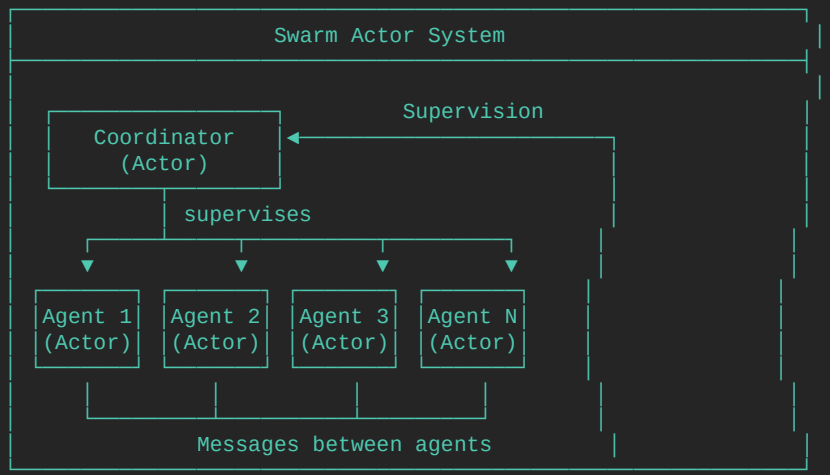
❌ Deadlock prevention — logical deadlocks все ще можливі:
A чекає на B, B чекає на A (через ask/request-reply)

Рішення: використовувати там, де переваги > недоліків

❌ Back-pressure — потрібна ручна реалізація

```
// Agent = Actor
struct DroneAgent {
    id: AgentId,
    position: Position,
    state: AgentState,
    battery: u8,
    mission: Option<Mission>,
}

// Agent messages
enum DroneMessage {
    MoveTo(Position),
    Scan(Area),
    Report { to: ActorRef<Coordinator> },
    Shutdown,
}
```

# MAC: Swarm як Actor System



```rust
// Coordinator -> Agent: Команди
enum CoordinatorToAgent {
    AssignMission(Mission),
    MoveTo(Position),
    ReturnToBase,
    Shutdown,
}

// Agent -> Coordinator: Звіти
enum AgentToCoordinator {
    StatusUpdate { agent: AgentId, status: AgentStatus },
    TargetDetected { agent: AgentId, target: Target },
    MissionComplete { agent: AgentId, mission: MissionId },
    LowBattery { agent: AgentId, level: u8 },
}

// Agent -> Agent: Peer communication
enum AgentToAgent {
    ShareTarget(Target),
    RequestAssistance(Position),
    FormationUpdate(FormationPosition),
}

// Event Bus (broadcast)
enum SwarmEvent {
    AlertLevel(AlertLevel),
    GlobalShutdown,
}
```

# Actor Model в екосистемі Rust

| Бібліотека | Опис | Статус |
|---|---|---|
| Manual (Tokio) | Власна реалізація на channels | Recommended |
| Actix | Потужний actor framework | Mature, complex |
| Ractor | Erlang-style actors | Modern, simple |
| xtra | Lightweight actors | Simple API |
| bastion | Fault-tolerant runtime | Supervision focus |
| coerce | Distributed actors | Remote actors |

Рекомендація для навчання:

1. Manual implementation на Tokio — розуміння концепцій
2. Ractor — простий Erlang-style API
3. Actix — для production web services

# Коли використовувати Actor Model?

✅ Використовуйте коли:
- Багато незалежних сутностей зі станом
- Потрібна ізоляція помилок (fault tolerance)
- Distributed система (location transparency)
- Event-driven архітектура
- 🤖 Мультиагентні системи!
- Game entities (NPCs, players)
- IoT devices
- Chat/messaging системи

❌ НЕ використовуйте коли:
- Shared state критичний для performance
- Tight synchronization потрібна
- Simple request-response API
- Low-latency requirements
- Багато мілісекундний overhead неприйнятний
- Проста CRUD application

Правило: якщо думаєте про систему як про "акторів" — використовуйте Actor Model

# Підсумок: Частина 1

Actor Model — модель конкурентних обчислень:
• Actor = state + mailbox + behavior
• Три аксіоми: SEND, CREATE, DESIGNATE
• Асинхронне message passing
• Ізольований стан — no race conditions

Порівняння:
• vs OOP: async messages замість sync calls
• vs Threads: lightweight, no locks
• vs CSP: named actors замість anonymous channels

Переваги: isolation, scalability, fault tolerance
Недоліки: overhead, debugging, learning curve

🤖 MAC: Agent = Actor — природна відповідність!

→ Частина 2: Реалізація на Tokio, supervision, патерни

Лекція 20 (продовження)

# Actor Model: Реалізація

## Практична реалізація на Tokio

Handle • Supervision • Stash • Timers • Patterns

🤖 Production-ready Agent Actors для MAC

Частина 2: Реалізація та патерни

О. С. Бичков • 2025

# План лекції (Частина 2)

1. Actor на Tokio — базова структура
2. ActorHandle pattern
3. Typed messages
4. Request-Reply pattern
5. Actor з таймерами
6. Stash pattern
7. Become pattern
8. Supervision strategies
9. Supervisor Actor
10. Actor Registry
11. 🤖 Agent Actor повністю
12. 🤖 Coordinator Actor
13. 🤖 Supervision hierarchy
14. 🤖 Full swarm system
15. Testing actors
16. Best practices

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

/// Actor = struct + mailbox receiver + run loop
pub struct MyActor {
    // Private state
    counter: i32,
    name: String,

    // Mailbox
    receiver: mpsc::Receiver<MyActorMessage>,
}

/// Messages
pub enum MyActorMessage {
    Increment,
    Decrement,
    GetValue { reply: oneshot::Sender<i32> },
}

impl MyActor {
    pub fn new(receiver: mpsc::Receiver<MyActorMessage>) -> Self {
        MyActor {
            counter: 0,
            name: "MyActor".to_string(),
            receiver,
        }
    }

    /// Main actor loop
    pub async fn run(mut self) {
```

```
impl MyActor {
    fn handle_message(&mut self, msg: MyActorMessage) {
        match msg {
            MyActorMessage::Increment => {
                self.counter += 1;
                println!("Counter: {}", self.counter);
            }

            MyActorMessage::Decrement => {
                self.counter -= 1;
                println!("Counter: {}", self.counter);
            }

            MyActorMessage::GetValue { reply } => {
                // Надсилаємо відповідь через oneshot channel
                let _ = reply.send(self.counter);
            }
        }
    }
}

// Запуск актора:
let (tx, rx) = mpsc::channel(100);
let actor = MyActor::new(rx);
tokio::spawn(actor.run());

// Надсилання повідомлень:
tx.send(MyActorMessage::Increment).await.unwrap();
```

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

/// Handle — публічний API для взаємодії з актором
#[derive(Clone)]
pub struct MyActorHandle {
    sender: mpsc::Sender<MyActorMessage>,
}

impl MyActorHandle {
    /// Створює актора і повертає handle
    pub fn new() -> Self {
        let (sender, receiver) = mpsc::channel(100);
        let actor = MyActor::new(receiver);

        // Spawn actor task
        tokio::spawn(actor.run());

        MyActorHandle { sender }
    }

    /// Fire-and-forget
    pub async fn increment(&self) -> Result<(), SendError> {
        self.sender.send(MyActorMessage::Increment).await
    }

    /// Request-Reply
    pub async fn get_value(&self) -> Result<i32, ActorError> {
        let (tx, rx) = oneshot::channel();
        self.sender.send(MyActorMessage::GetValue { reply: tx }).await?;
        rx.await.map_err(|_| ActorError::ActorDied)
```

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

/// Кожна операція — окремий message з типізованою відповіддю
pub enum AgentMessage {
    // Commands (no response)
    MoveTo(Position),
    StartPatrol { area: Area },
    Stop,

    // Queries (with typed response)
    GetPosition {
        reply: oneshot::Sender<Position>,
    },
    GetStatus {
        reply: oneshot::Sender<AgentStatus>,
    },
    GetBattery {
        reply: oneshot::Sender<u8>,
    },

    // Complex operations
    ExecuteMission {
        mission: Mission,
        reply: oneshot::Sender<Result<MissionResult, MissionError>>,
    },
}

// Handle методи інкапсулюють message creation
impl AgentHandle {
    pub async fn get_position(&self) -> Result<Position, ActorError> {
```

```rust
use tokio::time::{timeout, Duration};

impl AgentHandle {
    /// Generic request-reply з timeout
    async fn request<R>(
        &self,
        msg_fn: impl FnOnce(oneshot::Sender<R>) -> AgentMessage,
        timeout_duration: Duration,
    ) -> Result<R, ActorError> {
        let (tx, rx) = oneshot::channel();

        // Надсилаємо message
        self.sender.send(msg_fn(tx)).await
            .map_err(|_| ActorError::MailboxFull)?;

        // Чекаємо відповідь з timeout
        timeout(timeout_duration, rx).await
            .map_err(|_| ActorError::Timeout)?
            .map_err(|_| ActorError::ActorDied)
    }

    pub async fn get_status(&self) -> Result<AgentStatus, ActorError> {
        self.request(
            |reply| AgentMessage::GetStatus { reply },
            Duration::from_secs(5),
        ).await
    }
}

// Помилки
```

```rust
use tokio::time::{interval, Duration};
use tokio::select;

impl Agent {
    pub async fn run(mut self) {
        // Періодичний tick
        let mut tick = interval(Duration::from_millis(100));

        loop {
            select! {
                // Повідомлення мають пріоритет
                Some(msg) = self.receiver.recv() => {
                    if self.handle_message(msg).await.is_break() {
                        break;
                    }
                }

                // Періодичний tick
                _ = tick.tick() => {
                    self.on_tick().await;
                }
            }
        }
    }

    async fn on_tick(&mut self) {
        // Оновлення стану
        self.update_position();
        self.check_battery();
        self.scan_environment();
```

```rust
use std::collections::VecDeque;

struct Agent {
    state: AgentState,
    stash: VecDeque<AgentMessage>,
    receiver: mpsc::Receiver<AgentMessage>,
}

enum AgentState {
    Initializing,
    Ready,
    Busy,
}

impl Agent {
    async fn handle_message(&mut self, msg: AgentMessage) {
        match (&self.state, &msg) {
            // В стані Initializing — stash все крім Init
            (AgentState::Initializing, AgentMessage::Initialize(config)) => {
                self.initialize(config).await;
                self.state = AgentState::Ready;
                self.unstash_all().await;  // Обробити stashed
            }
            (AgentState::Initializing, _) => {
                self.stash.push_back(msg);  // Відкласти
            }

            // В стані Ready — обробляємо нормально
            (AgentState::Ready, _) => {
                self.process(msg).await;
```

```
/// Behavior — функція обробки повідомлень
type Behavior = Box<dyn FnMut(&mut Agent, AgentMessage) -> Option<Behavior>>;

struct Agent {
    behavior: Behavior,
    // ...
}

impl Agent {
    fn idle_behavior() -> Behavior {
        Box::new(|agent, msg| {
            match msg {
                AgentMessage::StartMission(m) => {
                    agent.current_mission = Some(m);
                    Some(Agent::mission_behavior())  // BECOME mission
                }
                _ => None,  // Keep current behavior
            }
        })
    }

    fn mission_behavior() -> Behavior {
        Box::new(|agent, msg| {
            match msg {
                AgentMessage::MissionComplete => {
                    agent.current_mission = None;
                    Some(Agent::idle_behavior())  // BECOME idle
                }
                AgentMessage::Abort => {
                    Some(Agent::returning_behavior())  // BECOME returning
```

```
enum SupervisionStrategy {
    OneForOne { max_restarts: u32, within: Duration },
    OneForAll { max_restarts: u32, within: Duration },
    RestForOne { max_restarts: u32, within: Duration },
}

enum SupervisionDecision {
    Restart,
    Stop,
    Escalate,  // Передати проблему вище
}
```

4️⃣ Simple-One-for-One
Для динамічних children однакового типу

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

struct Supervisor {
    children: JoinSet<Result<(), ActorError>>,
    child_specs: HashMap<ActorId, ChildSpec>,
    strategy: SupervisionStrategy,
    restart_counts: HashMap<ActorId, RestartHistory>,
}

impl Supervisor {
    async fn run(mut self) {
        loop {
            // Чекаємо на завершення будь-якого child
            if let Some(result) = self.children.join_next().await {
                match result {
                    Ok(Ok(())) => {
                        // Child завершився нормально
                    }
                    Ok(Err(error)) => {
                        // Child повернув помилку
                        self.handle_child_failure(error).await;
                    }
                    Err(join_error) => {
                        // Child panicked
                        self.handle_child_panic(join_error).await;
                    }
                }
            }
        }
    }
```

```rust
use std::collections::HashMap;
use tokio::sync::RwLock;
use std::sync::Arc;

/// Глобальний реєстр акторів
pub struct ActorRegistry {
    actors: RwLock<HashMap<ActorId, ActorEntry>>,
}

struct ActorEntry {
    handle: Box<dyn Any + Send + Sync>,
    actor_type: &'static str,
}

impl ActorRegistry {
    pub async fn register<H: Send + Sync + 'static>(
        &self, id: ActorId, handle: H
    ) {
        self.actors.write().await.insert(id, ActorEntry {
            handle: Box::new(handle),
            actor_type: std::any::type_name::<H>(),
        });
    }

    pub async fn get<H: Clone + 'static>(&self, id: ActorId) -> Option<H> {
        self.actors.read().await
            .get(&id)
            .and_then(|e| e.handle.downcast_ref::<H>())
            .cloned()
    }
```

```rust
pub struct AgentActor {
    id: AgentId,
    position: Position,
    state: AgentState,
    battery: u8,
    config: AgentConfig,
    receiver: mpsc::Receiver<AgentMessage>,
    coordinator: CoordinatorHandle,
}

impl AgentActor {
    pub async fn run(mut self) {
        println!("[Agent {}] Started", self.id);
        let mut tick = interval(Duration::from_millis(100));

        loop {
            select! {
                biased;
                Some(msg) = self.receiver.recv() => {
                    match self.handle_message(msg).await {
                        ControlFlow::Break(()) => break,
                        ControlFlow::Continue(()) => {}
                    }
                }
                _ = tick.tick() => {
                    self.tick().await;
                }
            }
        }
```

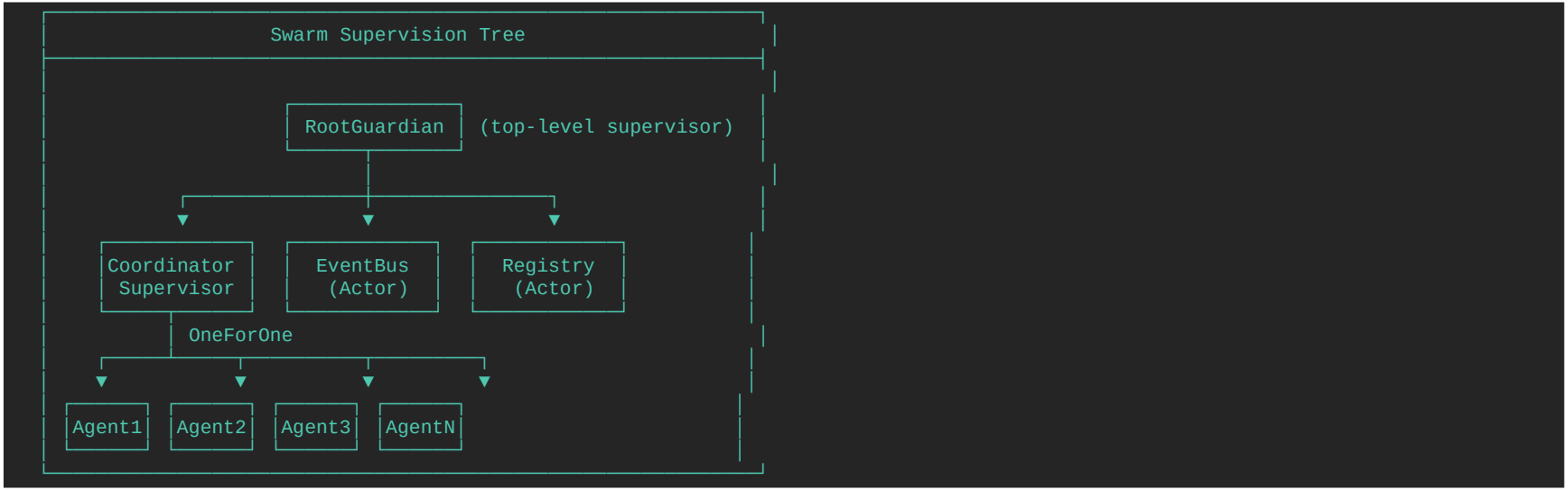
## 🕹️ MAC Agent Handle — ergonomic API

```rust
#[derive(Clone)]
pub struct AgentHandle {
    id: AgentId,
    sender: mpsc::Sender<AgentMessage>,
}

impl AgentHandle {
    pub fn spawn(id: AgentId, config: AgentConfig, coord: CoordinatorHandle) -> Self {
        let (tx, rx) = mpsc::channel(100);
        let actor = AgentActor::new(id, config, rx, coord);
        tokio::spawn(actor.run());
        AgentHandle { id, sender: tx }
    }

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

    pub async fn move_to(&self, pos: Position) -> Result<(), ActorError> {
        self.sender.send(AgentMessage::MoveTo(pos)).await
            .map_err(|_| ActorError::ActorDied)
    }

    pub async fn get_status(&self) -> Result<AgentStatus, ActorError> {
        let (tx, rx) = oneshot::channel();
        self.sender.send(AgentMessage::GetStatus { reply: tx }).await?;
        timeout(Duration::from_secs(5), rx).await??
    }

    pub async fn shutdown(&self) -> Result<(), ActorError> {
        self.sender.send(AgentMessage::Shutdown).await
            .map_err(|_| ActorError::ActorDied)
```

## MAC Coordinator Actor

```rust
pub struct CoordinatorActor {
    agents: HashMap<AgentId, AgentHandle>,
    missions: HashMap<MissionId, Mission>,
    receiver: mpsc::Receiver<CoordinatorMessage>,
    event_bus: broadcast::Sender<SwarmEvent>,
}

impl CoordinatorActor {
    pub async fn run(mut self) {
        println!("[Coordinator] Started");

        while let Some(msg) = self.receiver.recv().await {
            match msg {
                CoordinatorMessage::RegisterAgent { id, handle, reply } => {
                    self.agents.insert(id, handle);
                    let _ = reply.send(Ok(()));
                }

                CoordinatorMessage::AgentReport { agent_id, report } => {
                    self.handle_agent_report(agent_id, report).await;
                }

                CoordinatorMessage::AssignMission { mission } => {
                    self.assign_mission(mission).await;
                }

                CoordinatorMessage::Shutdown => {
                    self.shutdown_all().await;
                    break;
                }
```

# MAC: Supervision Hierarchy



```rust
#[tokio::main]
async fn main() -> Result<(), Box<dyn Error>> {
    // 1. Створюємо інфраструктуру
    let registry = Arc::new(ActorRegistry::new());
    let (event_tx, _) = broadcast::channel(1000);

    // 2. Запускаємо Coordinator
    let coordinator = CoordinatorHandle::spawn(event_tx.clone());
    registry.register("coordinator", coordinator.clone()).await;

    // 3. Запускаємо агентів
    for i in 0..10 {
        let agent = AgentHandle::spawn(
            AgentId(i),
            AgentConfig::default(),
            coordinator.clone(),
        );
        coordinator.register_agent(agent.id(), agent.clone()).await?;
        registry.register(format!("agent_{}", i), agent).await;
    }

    // 4. Graceful shutdown on Ctrl+C
    tokio::signal::ctrl_c().await?;

    coordinator.shutdown().await?;

    Ok(())
}
```

```rust
#[tokio::test]
async fn test_agent_actor() {
    // Створюємо mock coordinator
    let (coord_tx, mut coord_rx) = mpsc::channel(100);
    let mock_coordinator = MockCoordinatorHandle::new(coord_tx);

    // Створюємо агента
    let agent = AgentHandle::spawn(
        AgentId(1),
        AgentConfig::default(),
        mock_coordinator,
    );

    // Тест: команда MoveTo
    agent.move_to(Position::new(10.0, 20.0)).await.unwrap();

    // Перевіряємо статус
    let status = agent.get_status().await.unwrap();
    assert!(matches!(status.state, AgentState::Moving { .. }));

    // Тест: Coordinator отримує report
    tokio::time::sleep(Duration::from_millis(200)).await;
    let report = coord_rx.recv().await.unwrap();
    assert!(matches!(report, CoordinatorMessage::AgentReport { .. }));

    // Cleanup
    agent.shutdown().await.unwrap();
}
```

# Best Practices: Actors

✅ DO:
- Один actor — одна відповідальність
- Immutable messages
- Handle pattern для публічного API
- Timeout для request-reply
- Supervision для fault tolerance
- Registry для discovery
- Graceful shutdown

❌ DON'T:
- Blocking operations в actor loop
- Shared mutable state між actors
- Circular dependencies
- Unbounded mailboxes в production
- Sync request-reply chains (deadlock risk)

🤖 MAC: кожен Agent = Actor з supervisor

# Performance Tips

Mailbox sizing:
• Bounded для backpressure
• Розмір = expected_rate × processing_time
• Monitor mailbox length для alerts

Message passing:
• Clone messages якщо потрібно broadcast
• Arc<Message> для великих повідомлень
• Batch processing де можливо

Actor granularity:
• Не занадто дрібні (overhead)
• Не занадто великі (no parallelism)
• Один actor на логічну сутність

Scaling:
• Shard actors по ключу
• Pool actors для stateless operations
• Router pattern для load balancing

# Підсумок лекції

Реалізація Actor на Tokio:
- Actor = struct + mpsc receiver + run loop
- Handle pattern — публічний API
- Request-Reply з oneshot + timeout

Advanced patterns:
- Timers — periodic tick
- Stash — відкладання повідомлень
- Become — зміна поведінки
- Supervision — обробка помилок

🤖 MAC Architecture:
- Agent = Actor з state machine
- Coordinator = Actor supervisor
- Registry для discovery
- Event Bus для broadcast

→ Наступна лекція: Actor Model на Tokio — практикум

# Завдання для самостійної роботи

1. Counter Actor:
   - Increment, Decrement, GetValue
   - Handle 3 typed methods
   - Unit tests

2. Timer Actor:
   - Periodic tick (100ms)
   - Broadcast tick count
   - Shutdown on command

3. Agent Actor:
   - States: Idle, Moving, Scanning
   - Report to Coordinator
   - Stash commands while Busy

4. Supervisor:
   - OneForOne strategy
   - Max 3 restarts per minute
   - Escalate if exceeded

5. Full System:
   - Coordinator + 5 Agents
   - Registry
   - Graceful shutdown
   - Integration tests

# Actor Model опановано!

Actor • Handle • Supervision • Registry

Питання?