Підсумкова лекція

Інтеграція технологій Rust у МАС

Від базових структур даних до Production-системи

Ця лекція показує, як 20 технологій Rust
об'єднуються в єдину архітектуру рою БПЛА.

Кожна технологія вирішує конкретну проблему МАС.

Частина 1 з 4: Структури даних (Лекції 4-9)

О. С. Бичков • 2025

Філософія інтеграції технологій

Кожна технологія Rust вирішує конкретну проблему MAC:

Мультиагентна система — це не просто "багато агентів". Це складна екосистема, де потрібно вирішити десятки інженерних проблем:

- Як зберігати динамічні дані агента?
- Як швидко знаходити інформацію про світ?
- Як обробляти ситуації, коли дані відсутні?
- Як обробляти помилки без паніки системи?
- Як створити спільний інтерфейс для різних типів агентів?
- Як писати код, що працює з будь-яким типом агента?

Rust надає інструмент для кожної з цих проблем.
Наша задача — зрозуміти, КОЛИ і ЧОМУ використовувати кожен.

Карта технологій: від проблеми до рішення

Проблема MAC	Технологія	Лекція	Приклад у рої
Динамічні колекції	Vec<T>	4	Історія позицій дрона
Швидкий пошук за ключем	HashMap<K,V>	5	Реєстр відомих цілей
Відсутність даних	Option<T>	6	Ціль може бути не знайдена
Операції що можуть провалитись	Result<T,E>	7	Команда може не виконатись
Спільний інтерфейс типів	Traits	8	trait Agent для всіх дронів
Код для будь-якого типу	Generics	9	Registry<A: Agent>

Важливо розуміти:

Кожна технологія — це не просто "синтаксис Rust".
Це відповідь на конкретне питання проектування системи.

Коли ви бачите HashMap у коді MAC — це не випадковість.
Це свідомий вибір для забезпечення O(1) пошуку цілей.

Vec<T>: Проблема динамічних даних

Проблема, яку вирішує Vec:

Агент БПЛА працює в динамічному середовищі. Він постійно:
• Записує свої позиції для аналізу траєкторії
• Накопичує список виявлених цілей
• Отримує нові waypoints від координатора
• Буферизує команди для виконання

Всі ці дані мають спільну властивість:
їх кількість НЕВІДОМА на етапі компіляції.

Чому саме Vec?

• Гарантований O(1) доступ за індексом
• Амортизований O(1) для push
• Дані розташовані послідовно в пам'яті (cache-friendly)
• Володіє своїми елементами (ownership)

Vec<T>: Застосування в агенті БПЛА

Типові сценарії використання Vec у дроні:

```
pub struct DroneAgent {
    /// Історія позицій — для аналізу траєкторії та виявлення аномалій
    /// Зберігаємо останні 1000 точок для ковзного вікна аналізу
    position_history: Vec<Position>,

    /// Waypoints поточної місії — координатор надсилає список точок,
    /// дрон послідовно відвідує кожну
    patrol_waypoints: Vec<Position>,

    /// Виявлені цілі — накопичуються під час сканування
    detected_targets: Vec<Target>,

    /// Черга команд — буферизація коли команди надходять швидше,
    /// ніж дрон встигає виконувати
    command_queue: Vec<Command>,
}
```

Ключові операції з Vec у контексті MAC:

- push() — додати нову позицію, ціль, команду
- get(index) — отримати конкретний waypoint
- iter() — обійти всі цілі для аналізу
- retain() — видалити застарілі записи
- len() — перевірити чи є ще waypoints

```rust
impl DroneAgent {
    pub fn next_command(&mut self) -> Option<Command> {
        // Спочатку критичні команди
        if let Some(idx) = self.command_queue.iter()
            .position(|c| c.priority == Priority::Critical) {
            return Some(self.command_queue.remove(idx));
        }
        // Потім звичайні (FIFO)
        if !self.command_queue.is_empty() {
            return Some(self.command_queue.remove(0));
        }
        None
    }
}
```

HashMap<K,V>: Проблема швидкого пошуку

Проблема, яку вирішує HashMap:

Агент має "знання про світ" — це основа BDI архітектури (Beliefs).
Ці знання організовані як пари "ключ → значення":

• TargetId → інформація про ціль
• AgentId → статус союзника
• Координати → тип місцевості

Критична вимога: ШВИДКИЙ пошук за ключем.

Коли дрон виявляє ціль з ID=42, він повинен МИТТЄВО
дізнатись, чи це нова ціль, чи вже відома.

Використання Vec для цього:
```
targets.iter().find(|t| t.id == 42)  →  O(n) — ЗАНАДТО ПОВІЛЬНО!
```

Використання HashMap:
```
targets.get(&42)  →  O(1) — МИТТЄВО!
```

```
pub struct BeliefBase {
    /// Відомі цілі: ID → повна інформація
    /// Швидкий пошук: "Що я знаю про ціль #42?"
    known_targets: HashMap<TargetId, TargetBelief>,

    /// Союзні агенти: ID → їх статус
    /// Швидкий пошук: "Де зараз дрон #7?"
    friendly_agents: HashMap<AgentId, AgentBelief>,

    /// Карта небезпечних зон: координати → рівень загрози
    /// Швидкий пошук: "Чи безпечна точка (10, 20)?"
    danger_zones: HashMap<(i32, i32), ThreatLevel>,
}
```

Чому HashMap, а не Vec?

Уявіть 10,000 відомих цілей:

- Vec: пошук цілі = O(10,000) операцій
- HashMap: пошук цілі = O(1) операція

При 60 перевірках на секунду це різниця між
600,000 операцій vs 60 операцій!

```rust
// ✅ Ефективно: один пошук через Entry API
beliefs.known_targets
    .entry(target_id)
    .and_modify(|existing| {
        existing.last_position = new_position;
        existing.last_seen = Instant::now();
        existing.confidence = 1.0;
    })
    .or_insert_with(|| TargetBelief::new(target_id, new_position));
```

Entry API — ідіоматичний спосіб роботи з HashMap у Rust.
Він гарантує один пошук замість двох.

Option<T>: Проблема відсутності даних

Проблема, яку вирішує Option:

Агент діє в НЕВИЗНАЧЕНОМУ середовищі. Багато операцій
можуть не мати результату:

- Сенсор сканує область — цілі може НЕ БУТИ
- Шукаємо найближчого союзника — союзників може НЕ БУТИ
- Запитуємо наступний waypoint — місія може ЗАВЕРШИТИСЬ
- Шукаємо маршрут до точки — маршрут може НЕ ІСНУВАТИ

У мовах як Java/C# це вирішується через null:

```
Target target = findTarget();  // може бути null
target.attack();  // NullPointerException!
```

Rust НЕ МАЄ null. Замість цього — Option<T>:

```
let target: Option<Target> = find_target();
// Компілятор ЗМУСИТЬ перевірити перед використанням!
```

```rust
impl DroneAgent {
    /// Знайти найближчу ціль — може не бути жодної!
    pub fn find_nearest_target(&self) -> Option<&Target> {
        self.detected_targets.iter()
            .min_by_key(|t| self.distance_to(&t.position) as i32)
    }
}

// Використання — компілятор ЗМУШУЄ обробити відсутність
match drone.find_nearest_target() {
    Some(target) => {
        println!("Знайдено ціль на {:?}", target.position);
        drone.engage(target);
    }
    None => {
        println!("Цілей не виявлено, продовжую патрулювання");
```

Перевага: неможливо "забути" перевірити на None!
Компілятор не дозволить використати Option<T> як T.

```
impl DroneAgent {
    /// Отримати позицію найближчого ворога (якщо є)
    pub fn nearest_enemy_position(&self) -> Option<Position> {
        self.beliefs.known_targets.values()
            .filter(|t| t.is_enemy)                    // Тільки вороги
            .min_by_key(|t| self.distance_to(&t.pos)) // Найближчий
            .map(|t| t.pos)                           // Витягуємо позицію
    }

    /// Атакувати найближчого ворога, якщо він є
    pub fn attack_nearest_if_possible(&mut self) {
        // if let — зручний патерн для Option
        if let Some(enemy_pos) = self.nearest_enemy_position() {
            self.set_attack_target(enemy_pos);
        }
```

Основні комбінатори: map(), and_then(), unwrap_or(), filter()
Вони дозволяють писати виразний код без вкладених match.

```
    /// Отримати ціль або використати позицію бази
    pub fn get_destination(&self) -> Position {
        self.current_target
            .unwrap_or(self.base_position)  // Default якщо None
    }
}
```

Result<T,E>: Проблема операцій що можуть провалитись

Проблема, яку вирішує Result:

На відміну від Option (є/немає), Result описує операції,
що можуть ПРОВАЛИТИСЬ з конкретною ПРИЧИНОЮ:

- Команда руху — батарея може бути розряджена
- Атака цілі — зброя може бути не заряджена
- Зв'язок з координатором — канал може бути зайнятий
- Читання сенсора — сенсор може бути пошкоджений

Різниця від Option:
- Option: "Результату немає" (нормальна ситуація)
- Result: "Операція провалилась" (помилка з причиною)

Result<T, E>:
- Ok(value) — операція успішна, ось результат
- Err(error) — операція провалилась, ось причина

```rust
/// Помилки операцій дрона
#[derive(Debug, thiserror::Error)]
pub enum DroneError {
    #[error("Батарея нижче мінімуму: {0}%")]
    LowBattery(u8),

    #[error("Точка {0:?} знаходиться в забороненій зоні")]
    NoFlyZone(Position),

    #[error("Ціль {0:?} недосяжна з поточної позиції")]
    TargetUnreachable(TargetId),

    #[error("Зброя не заряджена, залишилось {0} набоїв")]
```

Кожен варіант помилки несе КОНТЕКСТ:

- LowBattery(15) — знаємо точний рівень
- NoFlyZone(pos) — знаємо яка саме точка заборонена
- TargetUnreachable(id) — знаємо яка ціль недосяжна

Це критично для діагностики та логування в MAC!

```rust
impl DroneAgent {
    /// Виконати складну місію з багатьма кроками
    pub fn execute_attack_mission(&mut self, target_id: TargetId) -> Result<(), DroneError> {
        // Кожен крок може провалитись — ? передасть помилку
        self.check_battery()?;           // LowBattery?
        self.check_weapon_ready()?;      // WeaponNotReady?

        let target = self.beliefs
            .get_target(&target_id)
            .ok_or(DroneError::TargetUnreachable(target_id))?;

        self.fly_to(target.position)?;   // NoFlyZone?
        self.engage_target(target)?;     // Різні помилки?

        Ok(())  // Все успішно!
    }
}

// Виклик
match drone.execute_attack_mission(target_id) {
    Ok(()) => log::info!("Місія виконана успішно"),
    Err(DroneError::LowBattery(level)) => {
        log::warn!("Місія перервана: батарея {}%", level);
        drone.return_to_base();
    }
    Err(e) => log::error!("Місія провалена: {}", e),
}
```

Traits: Проблема спільного інтерфейсу

Проблема, яку вирішують Traits:

У рої є РІЗНІ типи агентів:
• ScoutDrone — розвідник з камерою
• CombatDrone — штурмовик зі зброєю
• RelayDrone — ретранслятор зв'язку
• CommandDrone — командир групи

Але всі вони мають СПІЛЬНІ операції:
• Всі мають позицію
• Всі можуть рухатись
• Всі отримують повідомлення
• Всі оновлюються кожен тік

Проблема: як написати код, що працює з БУДЬ-ЯКИМ дроном?

Відповідь: визначити СПІЛЬНИЙ ІНТЕРФЕЙС через trait.

```rust
/// Базовий trait для ВСІХ агентів у системі
pub trait Agent: Send + Sync {
    /// Унікальний ідентифікатор агента
    fn id(&self) -> AgentId;

    /// Поточна позиція в просторі
    fn position(&self) -> Position;

    /// Оновлення стану (викликається кожен тік)
    fn update(&mut self, delta_time: f32);

    /// Обробка вхідного повідомлення
    fn receive_message(&mut self, msg: AgentMessage);
}
```

Send + Sync — маркерні traits що дозволяють передавати між потоками. Обов'язкові для багатопотокової MAC!

```rust
    fn scan(&self, radius: f32) -> Vec<Detection>;
    fn sensor_range(&self) -> f32;
}

/// Додатковий trait для озброєних агентів
pub trait Armed: Agent {
    fn fire(&mut self, target: TargetId) -> Result<(), WeaponError>;
    fn ammo(&self) -> u32;
}
```

```rust
pub struct ScoutDrone {
    id: AgentId,
    position: Position,
    camera: Camera,
}

// ScoutDrone реалізує Agent (базовий) + Sensing (сканування)
impl Agent for ScoutDrone {
    fn id(&self) -> AgentId { self.id }
    fn position(&self) -> Position { self.position }
    fn update(&mut self, dt: f32) { /* логіка руху */ }
    fn receive_message(&mut self, msg: AgentMessage) { /* обробка */ }
}

impl Sensing for ScoutDrone {
    fn scan(&self, radius: f32) -> Vec<Detection> {
        self.camera.capture_in_radius(self.position, radius)
    }
    fn sensor_range(&self) -> f32 { self.camera.max_range }
}

// CombatDrone реалізує Agent + Armed (але НЕ Sensing)
impl Agent for CombatDrone { /* ... */ }
impl Armed for CombatDrone {
    fn fire(&mut self, target: TargetId) -> Result<(), WeaponError> { /* ... */ }
    fn ammo(&self) -> u32 { self.weapon.ammo_count }
}
```

```rust
/// Статичний поліморфізм (compile-time, generics)
/// Швидкий, але один тип за раз
fn find_nearest<A: Agent>(agents: &[A], target: Position) -> Option<&A> {
    agents.iter()
        .min_by_key(|a| a.position().distance_to(&target) as i32)
}

/// Динамічний поліморфізм (runtime, trait objects)
/// Гнучкий, можна змішувати різні типи
fn broadcast_to_all(agents: &mut [Box<dyn Agent>], msg: AgentMessage) {
    for agent in agents {
        agent.receive_message(msg.clone());
    }
}
```

dyn Agent — trait object для гетерогенних колекцій.
A: Agent + Sensing — trait bounds для обмеження типів.

```rust
    scanners.iter()
        .flat_map(|s| s.scan(s.sensor_range()))
        .collect()
}
```

Generics: Проблема дублювання коду

Проблема, яку вирішують Generics:

Уявіть, що потрібен реєстр агентів. Без generics:

```
struct ScoutRegistry { agents: HashMap<AgentId, ScoutDrone> }
struct CombatRegistry { agents: HashMap<AgentId, CombatDrone> }
struct RelayRegistry { agents: HashMap<AgentId, RelayDrone> }
```

Кожен має ІДЕНТИЧНІ методи:

```
fn register(&mut self, agent)
fn get(&self, id) -> Option<&Agent>
fn update_all(&mut self, dt: f32)
```

Це дублювання коду! При зміні логіки — змінювати всюди.

Рішення: Generic структура AgentRegistry<A>
Один код працює для БУДЬ-ЯКОГО типу агента.

```rust
/// Реєстр для БУДЬ-ЯКОГО типу агента
pub struct AgentRegistry<A: Agent> {
    agents: HashMap<AgentId, A>,
}

impl<A: Agent> AgentRegistry<A> {
    pub fn new() -> Self {
        Self { agents: HashMap::new() }
    }

    pub fn register(&mut self, agent: A) {
        self.agents.insert(agent.id(), agent);
    }

    pub fn get(&self, id: &AgentId) -> Option<&A> {
        self.agents.get(id)
    }

    pub fn update_all(&mut self, dt: f32) {
        for agent in self.agents.values_mut() {
            agent.update(dt);
        }
    }

    pub fn broadcast(&mut self, msg: AgentMessage) {
        for agent in self.agents.values_mut() {
            agent.receive_message(msg.clone());
        }
    }
}
```

```rust
/// Тільки агенти з сенсорами можуть використовуватись тут
pub fn collective_scan<A: Agent + Sensing>(
    registry: &AgentRegistry<A>,
    center: Position,
    radius: f32,
) -> Vec<Detection> {
    registry.agents.values()
        .filter(|a| a.position().distance_to(&center) < radius)
        .flat_map(|a| a.scan(a.sensor_range()))
        .collect()
}

/// Множинні bounds через where (для читабельності)
pub fn attack_coordinator<A>(registry: &mut AgentRegistry<A>, target: TargetId)
where
    A: Agent + Armed + Sensing — тип A має реалізувати ВСІ три traits.
Це гарантується на етапі компіляції!
    for agent in registry.agents.values_mut() {
        if agent.scan(100.0).iter().any(|d| d.target_id == target) {
            let _ = agent.fire(target);
        }
    }
}
```

Підсумок: Структури даних для MAC

Лекції 4-9 дали фундамент для моделювання агента:

Vec<T> — динамічні колекції
→ Історія, waypoints, черги команд
→ O(1) доступ, послідовна пам'ять

HashMap<K,V> — асоціативні масиви
→ BeliefBase, реєстр цілей, карта
→ O(1) пошук за ключем

Option<T> — наявність/відсутність
→ "Ціль не знайдена" — це норма, не помилка
→ Компілятор змушує обробити None

Result<T,E> — успіх/провал з причиною
→ "Команда провалилась через низьку батарею"
→ Оператор ? для елегантної пропагації

Traits — спільний інтерфейс
→ Agent, Sensing, Armed для різних типів дронів

Generics — універсальний код
→ AgentRegistry<A> працює з будь-яким агентом

Підсумкова лекція

Smart Pointers та керування пам'яттю

Box • Rc • Arc • RefCell • Lifetimes

Як керувати складними структурами даних
та спільним доступом у МАС?

Smart Pointers — відповідь Rust на ці питання.

Частина 2 з 4: Керування пам'яттю (Лекції 10-13)

О. С. Бичков • 2025

Проблеми керування пам'яттю в MAC

Мультиагентна система створює унікальні виклики для пам'яті:

1. Рекурсивні структури даних
   Дерева рішень, ієрархічні плани — структура посилається сама на себе.
   Rust не знає розмір на етапі компіляції → потрібен Box.

2. Спільні дані між агентами
   Карта світу, конфігурація — багато агентів читають одні дані.
   Кожен агент не може "володіти" картою → потрібен Rc/Arc.

3. Мутація через спільні посилання
   Спільний стан місії — агенти мають змінювати його.
   Порушує правила borrowing → потрібен RefCell.

4. Посилання на частини структур
   View на beliefs агента без копіювання.
   Як гарантувати що дані живуть достатньо? → Lifetimes.

Box<T>: Проблема рекурсивних структур

Проблема: структура що містить саму себе

BDI агент має плани, де крок плану може містити підплан:

```
PlanStep::Sequence(vec![
    PlanStep::Action(fly),
    PlanStep::IfThen {
        condition: ...,
        then_step: PlanStep::Sequence(...),  // ← Рекурсія!
        else_step: PlanStep::Action(...),
    },
])
```

Чому це проблема для Rust?

Компілятор має знати РОЗМІР кожного типу на етапі компіляції.
Але розмір PlanStep залежить від того, скільки рівнів вкладеності...
А це може бути нескінченно! → Компілятор не може обчислити розмір.

```rust
/// Крок плану BDI агента
pub enum PlanStep {
    /// Примітивна дія (атомарна)
    Action(Action),

    /// Послідовність кроків
    Sequence(Vec<PlanStep>),  // Vec вже на heap, ОК

    /// Умовне виконання — тут потрібен Box!
    IfThen {
        condition: Condition,
        then_step: Box<PlanStep>,  // Box<T> має фіксований розмір!
        else_step: Option<Box<PlanStep>>,
    },
    body: Box<PlanStep>, // Рекурсія через Box
    },
}
```

Тепер компілятор знає розмір:

- Action — фіксований
- Sequence — Vec (вказівник + len + capacity)
- IfThen — Condition + Box (вказівник) + Option<Box> (вказівник)

```
/// Створення плану патрулювання для BDI агента
fn create_patrol_plan() -> PlanStep {
    PlanStep::Sequence(vec![
        // 1. Злетіти
        PlanStep::Action(Action::TakeOff),

        // 2. Поки є waypoints — обходити їх
        PlanStep::While {
            condition: Condition::HasWaypoints,
            body: Box::new(PlanStep::Sequence(vec![  // Box для рекурсії
                // Летіти до наступного waypoint
                PlanStep::Action(Action::FlyToNextWaypoint),

                // Сканувати область
                PlanStep::Action(Action::Scan { radius: 100.0 }),

                // Якщо знайдено ціль — доповісти
                PlanStep::IfThen {
                    condition: Condition::TargetDetected,
                    then_step: Box::new(  // Ще один Box
                        PlanStep::Action(Action::ReportTarget)
                    ),
                    else_step: None,
                },
            ])),
        },

        // 3. Повернутись на базу
        PlanStep::Action(Action::ReturnToBase),
    ])
```

Rc<T>: Проблема спільного володіння

Проблема: кілька власників одних даних

У Rust кожне значення має ОДНОГО власника (ownership).
Але в MAC часто потрібно, щоб КІЛЬКА агентів мали доступ до
ТИХ САМИХ даних:

• Карта світу — всі 100 дронів читають ту саму карту
• Конфігурація місії — всі агенти знають параметри
• Спільний стан місії — прогрес, знайдені цілі

Наївний підхід — копіювати:

```
let map_copy1 = world_map.clone();  // 10 MB
let map_copy2 = world_map.clone();  // ще 10 MB
// 100 агентів × 10 MB = 1 GB пам'яті!
```

Потрібен спосіб ДІЛИТИ дані без копіювання.

```rust
use std::rc::Rc;

// Карта світу — створюємо ОДИН раз
let world_map = Rc::new(WorldMap::load("map.json"));  // Лічильник = 1

// Кожен агент отримує свій Rc (дешевий clone — тільки +1 до лічильника)
let agent1_map = Rc::clone(&world_map);  // Лічильник = 2
let agent2_map = Rc::clone(&world_map);  // Лічильник = 3
let agent3_map = Rc::clone(&world_map);  // Лічильник = 4

// Всі чотири Rc посилаються на ТІ САМІ дані!
// world_map, agent1_map, agent2_map, agent3_map → [один WorldMap]

// Коли agent1_map виходить зі scope — лічильник = 3
// Коли всі виходять — лічильник = 0 → дані звільняються
```

Rc::clone() — O(1), копіює лише вказівник + інкремент лічильника.
Це НАБАГАТО дешевше ніж clone() самих даних!

```
use std::sync::Arc;
use std::thread;

// Спільна карта для багатопотокового рою
let world_map = Arc::new(WorldMap::load("map.json"));

// Запускаємо 100 агентів у різних потоках
let handles: Vec<_> = (0..100).map(|id| {
    let map = Arc::clone(&world_map);  // Кожен потік отримує Arc
    thread::spawn(move || {
        let agent = DroneAgent::new(id, map);  // Агент має доступ до карти
        agent.run();
    })
}).collect();
```

що безпечний для багатопотоковості.

Rc vs Arc: Коли що використовувати?

Критерій	Rc<T>	Arc<T>
Потокобезпечність	❌ Один потік	✅ Багато потоків
Продуктивність	Швидший	Трохи повільніший
Overhead	Звичайний лічильник	Атомарні операції
Traits	!Send, !Sync	Send, Sync
Use case	Однопотокові симуляції	Паралельні MAC

Правило вибору:

- Однопотоковий код (рання розробка, тести) → Rc<T>
- Багатопотоковий код (production MAC) → Arc<T>

У реальних MAC майже завжди потрібен Arc, бо агенти працюють у різних потоках або async задачах.

Arc трохи повільніший через атомарні операції, але ця різниця незначна порівняно з вартістю копіювання даних.

RefCell<T>: Проблема мутації через спільне посилання

Проблема: правила borrowing занадто суворі

Rust borrowing rules:
• Можна мати багато &T (іммутабельних посилань) АБО
• Один &mut T (мутабельне посилання), НЕ обидва одночасно

Але з Rc<T> ми маємо багато "власників" → тільки &T доступ!

```
let shared_state = Rc::new(MissionState::new());
let state_ref1 = Rc::clone(&shared_state);
let state_ref2 = Rc::clone(&shared_state);

// Як змінити state? Всі мають тільки &MissionState!
state_ref1.targets_found.push(target);  // ❌ Не можна!
```

Потрібен механізм для мутації "зсередини" іммутабельного посилання.
Це називається Interior Mutability.

```rust
use std::cell::RefCell;
use std::rc::Rc;

/// Спільний стан місії з interior mutability
pub struct SharedMissionState {
    targets_found: RefCell<Vec<TargetId>>,  // Можна змінювати!
    area_scanned: RefCell<f32>,
    mission_status: RefCell<MissionStatus>,
}

// Кілька агентів мають Rc на той самий стан
let mission = Rc::new(SharedMissionState::new());
let agent1_mission = Rc::clone(&mission);
let agent2_mission = Rc::clone(&mission);

// Агент 1 знаходить ціль і записує
agent1_mission.targets_found.borrow_mut().push(target_id);  // Runtime check!

// Агент 2 оновлює площу
*agent2_mission.area_scanned.borrow_mut() += 100.0;  // Runtime check!

// Читання
let count = agent1_mission.targets_found.borrow().len();
```

⚠️ RefCell — тільки для однопотокового коду!
Для багатопотоковості: Arc<Mutex<T>> замість Rc<RefCell<T>>

```
// ✅ Правильно: короткий borrow
{
    let mut data = state.borrow_mut();
    data.push(value);
}  // borrow закінчується тут
// Тепер можна знову borrow
```

```rust
use std::sync::{Arc, Mutex};

/// Спільний стан рою (thread-safe)
pub struct SwarmState {
    targets: Mutex<HashMap<TargetId, TargetInfo>>,
    stats: Mutex<SwarmStats>,
}

let shared = Arc::new(SwarmState::new());

// У кожному потоці/агенті:
let state = Arc::clone(&shared);
thread::spawn(move || {
    // Блокуємо для запису
    let mut targets = state.targets.lock().unwrap();
    targets.insert(id, info);
    // Оновлюємо статистику
    let mut stats = state.stats.lock().unwrap();
    stats.targets_found += 1;
});
```

⚠️ Небезпека deadlock! Якщо потік A тримає lock1 і чекає lock2,
а потік B тримає lock2 і чекає lock1 → обидва заблоковані назавжди.

```rust
use std::sync::{Arc, RwLock};

/// Карта світу — часто читається, рідко оновлюється
let world_map = Arc::new(RwLock::new(WorldMap::new()));

// У кожному агенті:
let map = Arc::clone(&world_map);

// Читання — багато одночасно
{
    let map_read = map.read().unwrap();  // Не блокує інших readers
    let cell = map_read.get_cell(x, y);
}  // read lock звільняється

// Запис — ексклюзивно
{
    let mut map_write = map.write().unwrap();  // Блокує всіх!
```

Коли що: Mutex для часто змінюваних даних, RwLock для read-heavy.

Lifetimes: Проблема посилань на частини структур

Проблема: створити view без копіювання

Уявіть, що система аналізу хоче переглянути beliefs агента:

```
fn analyze(beliefs: &BeliefBase) -> Analysis { ... }
```

Але BeliefBase великий! Аналізатору потрібна тільки частина:
• targets (для аналізу загроз)
• position (для оцінки дистанцій)

Lifetime — це відповідь Rust на питання:
"Як довго це посилання буде валідним?"

Lifetime annotation ('a) — спосіб сказати компілятору про зв'язок між часом життя різних посилань.

```rust
/// View на частину beliefs — без копіювання!
/// 'a означає: "ці посилання живуть стільки ж, скільки джерело"
pub struct BeliefView<'a> {
    targets: &'a HashMap<TargetId, TargetBelief>,
    position: &'a Position,
    battery: &'a u8,
}

impl DroneAgent {
    /// Створити легкий view для аналізу
    /// Lifetime 'a = час життя self (агента)
    pub fn belief_view(&self) -> BeliefView<'_> {
        BeliefView {
            targets: &self.beliefs.known_targets,
            position: &self.position,
            battery: &self.battery,
        }
    }
}

// Використання
let agent = DroneAgent::new();
let view = agent.belief_view();  // view позичає дані з agent

// view живе поки agent живе
analyze_threats(&view);

// Якщо agent буде переміщений/знищений — view стане невалідним
// Rust НЕ ДОЗВОЛИТЬ це скомпілювати!
```

'_ — anonymous lifetime, компілятор виводить автоматично.

Lifetimes гарантують: view ніколи не переживе agent.

```
/// Знайти найближчу загрозу з view
/// 'a зв'язує lifetime повернення з lifetime входу
fn find_nearest_threat<'a>(
    view: &'a BeliefView<'a>,
) -> Option<&'a TargetBelief> {
    view.targets.values()
        .filter(|t| t.threat_level >= ThreatLevel::High)
        .min_by_key(|t| t.position.distance_to(view.position) as i32)
}

// Компілятор розуміє:
// Повернуте &TargetBelief живе не довше ніж view
```

Lifetime elision rules — компілятор часто виводить lifetimes сам:

- Один вхідний параметр → його lifetime присвоюється виходу
- &self → lifetime self присвоюється виходу

Явні 'a потрібні коли є неоднозначність.

Загальна картина: комбінації Smart Pointers

Сценарій	Рішення	Приклад
Рекурсивна структура	Box<T>	Дерево планів
Спільне читання (1 потік)	Rc<T>	Спільна карта
Спільне читання (N потоків)	Arc<T>	Карта для рою
Спільний запис (1 потік)	Rc<RefCell<T>>	Стан місії
Спільний запис (N потоків)	Arc<Mutex<T>>	Статистика рою
Read-heavy (N потоків)	Arc<RwLock<T>>	World map
Посилання на частину	&'a T, Lifetimes	BeliefView<'a>

Ключове правило вибору:

1. Один потік: Rc<T>, Rc<RefCell<T>>
2. Багато потоків: Arc<T>, Arc<Mutex<T>>, Arc<RwLock<T>>
3. Краще уникати спільного мутабельного стану → Channels!

Підсумок: Smart Pointers для MAC

Лекції 10-13 дали інструменти керування пам'яттю:

Box<T> — heap allocation
→ Рекурсивні структури (плани, дерева рішень)
→ Фіксований розмір вказівника

Rc<T> / Arc<T> — спільне володіння
→ Багато агентів → одні дані (карта, конфіг)
→ Rc для одного потоку, Arc для багатьох

RefCell<T> / Mutex<T> — interior mutability
→ Мутація через спільне посилання
→ RefCell для одного потоку, Mutex для багатьох

RwLock<T> — оптимізація для read-heavy
→ Карта світу: часто читається, рідко пишеться

Lifetimes ('a) — зв'язування часу життя
→ View на частини даних без копіювання
→ Гарантія валідності посилань

→ Частина 3: Паралелізм та асинхронність

Підсумкова лекція

Паралелізм та асинхронність

Threads • Channels • async/await • Tokio • Streams

Як запустити 100 агентів паралельно?
Як забезпечити комунікацію без data races?
Як масштабуватись до 10,000+ агентів?

Частина 3 з 4: Паралелізм (Лекції 14-21)

О. С. Бичков • 2025

Еволюція моделі паралелізму в MAC

Як запускати багато агентів?

Етап 1: Послідовне виконання (просто, але повільно)
  for agent in agents { agent.update(); }
  Проблема: 100 агентів × 10ms = 1 секунда на тік!

Етап 2: Потоки (threads) — справжній паралелізм
  Кожен агент у своєму потоці → всі працюють одночасно
  Проблема: потік = 1-8 MB стека. 1000 агентів = 8 GB RAM!

Етап 3: Async/await — легкі задачі
  Кооперативна багатозадачність на кількох потоках
  1 async task ≈ 1-10 KB. 100,000 агентів = сотні MB.

У реальній MAC: комбінація — CPU-bound на threads,
I/O-bound на async.

Threads: Що таке потік?

Потік (thread) — незалежний шлях виконання в програмі.

Кожен потік має:
• Свій стек — локальні змінні ізольовані
• Свій Program Counter — де зараз виконується код
• Доступ до спільної пам'яті (heap) — тут потрібна синхронізація

Чому потоки для MAC?

Кожен агент — автономна сутність зі своїм циклом.
Природа агентів ПАРАЛЕЛЬНА:
Поки дрон А летить до waypoint,

Ключова гарантія Rust: data race НЕМОЖЛИВИЙ!
Компілятор перевіряє доступ до спільних даних.
Потоки реалізують цю паралельність на hardware:
на 8-ядерному CPU — 8 агентів справді працюють ОДНОЧАСНО.

```rust
use std::thread;

/// Запустити рій агентів у окремих потоках
fn spawn_fleet(count: usize) -> Vec<thread::JoinHandle<()>> {
    (0..count)
        .map(|id| {
            // spawn повертає JoinHandle для контролю потоку
            thread::spawn(move || {  // move переміщує id у потік
                println!("Дрон {} активовано", id);

                // Головний цикл агента
                loop {
                    sense();
                    think();
                    act();
```

Без move: closure позичає id через &
→ помилка: id не живе достатньо довго!
З move: closure володіє копією id → все працює.

```rust
        .collect()
}

// Чекаємо завершення всіх потоків
for handle in handles {
    handle.join().unwrap();  // join блокує до завершення потоку
}
```

Channels: Філософія Message Passing

Два способи комунікації між потоками:

1. Shared State (спільна пам'ять)
   Потоки читають/пишуть ту саму структуру.
   Потрібна синхронізація: Mutex, RwLock.
   Легко помилитись → deadlock, race conditions.

2. Message Passing (передача повідомлень)
   Потоки обмінюються через канали.
   Немає спільного стану → немає race conditions!
   Кожен потік "володіє" своїми даними.

Золоте правило Go (актуальне і для Rust):

"Do not communicate by sharing memory;
share memory by communicating."

Для MAC: channels — РЕКОМЕНДОВАНИЙ підхід.
Агенти обмінюються повідомленнями як у реальному світі!

```rust
use std::sync::mpsc;

// Створюємо канал: tx для відправки, rx для отримання
let (tx, rx) = mpsc::channel::<DroneReport>();

// Кожен агент отримує КЛОН відправника
for id in 0..100 {
    let agent_tx = tx.clone();  // clone() — дозволяє багато відправників
    thread::spawn(move || {
        loop {
            // Агент працює...
            if found_target {
                let report = DroneReport { drone_id: id, target_pos: pos };
                agent_tx.send(report).unwrap();  // Надсилаємо в канал
            }
```

tx.clone() — O(1), копіює лише вказівник + atomic increment.

Канал закривається коли BCI Sender знищені.

```rust
drop(tx);  // Важливо: закриваємо оригінал, лишаються тільки клони

// Координатор отримує BCI звіти через ОДИН rx
for report in rx {  // Ітерація до закриття каналу
    println!("Дрон {} знайшов ціль на {:?}", report.drone_id, report.target_pos);
    process_report(report);
}
```

```rust
/// Всі типи повідомлень агентів
#[derive(Debug, Clone)]
pub enum AgentMessage {
    TargetFound { drone_id: u32, target: Target },
    StatusUpdate { drone_id: u32, pos: Position, battery: u8 },
    BatteryLow { drone_id: u32, level: u8 },
    MissionComplete { drone_id: u32 },
    RequestBackup { drone_id: u32, urgency: u8 },
}

// Координатор ЗОБОВ'ЯЗАНИЙ обробити всі варіанти
for msg in rx {
    match msg {
        AgentMessage::TargetFound { drone_id, target } => {
            log::info!("Дрон {} виявив ціль", drone_id);
            assign_attack_mission(target);
        }
        AgentMessage::BatteryLow { drone_id, level } => {
            log::warn!("Дрон {} — критичний заряд {}%", drone_id, level);
            recall_drone(drone_id);
        }
        // Компілятор вимагає обробити ВСІ варіанти!
        _ => { /* інші */ }
    }
}
```

Async: Проблема масштабування потоків

Проблема: потоки НЕ масштабуються до тисяч агентів

Кожен потік споживає:
• 1-8 MB стека (залежить від ОС)
• Системні ресурси для scheduling
• CPU час на context switch

Для 10,000 агентів:
• Потоки: 10,000 × 2 MB = 20 GB RAM — НЕРЕАЛЬНО!
• Context switches: мільйони на секунду — CPU перевантажений

Рішення: async/await — кооперативна багатозадачність

Async task — НЕ потік. Це легка структура даних (~1-10 KB).
Замість "один потік на агента" — "тисячі задач на кілька потоків".

10,000 async tasks × 10 KB = 100 MB — РЕАЛЬНО!

Async: Кооперативна багатозадачність

Як працює async/await?

async fn — функція що може "призупинитись" і віддати управління.
.await — точка де функція призупиняється, чекаючи результат.

Ключова відмінність від потоків:

Потоки — ВИТІСНЯЮЧА (preemptive) багатозадачність:
  ОС вирішує коли перемикати потоки (примусово).
  Потік може бути перерваний будь-де.

Async — КООПЕРАТИВНА багатозадачність:
  Задача САМА вирішує коли віддати управління (.await).
  Між точками await — гарантовано атомарне виконання.

Перевага async:
• Легші перемикання (немає context switch ОС)
• Менше накладних витрат
• Більш передбачувана поведінка

```rust
// async fn повертає Future, НЕ виконується відразу!
async fn scan_area(radius: f32) -> Vec<Target> {
    // Цей код виконається тільки коли Future буде await
    tokio::time::sleep(Duration::from_secs(1)).await;  // Точка suspend
    vec![Target::new()]  // Результат
}

// Варіант 1: Створили Future, але НЕ запустили!
let future = scan_area(100.0);  // Нічого не відбувається!

// Варіант 2: await запускає виконання
let targets = scan_area(100.0).await;  // Тепер виконується!

// Варіант 3: spawn у окрему задачу
let handle = tokio::spawn(scan_area(100.0));  // Виконується паралельно
let targets = handle.await.unwrap();  // Чекаємо результат
```

Rust Futures — lazy! Вони виконуються тільки коли хтось "тягне".

```rust
#[tokio::main]  // Макрос ініціалізує runtime
async fn main() {
    // Тепер можна використовувати .await
}
```

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

#[tokio::main]
async fn main() {
    // Async канал (tokio версія)
    let (tx, mut rx) = mpsc::channel::<DroneReport>(1000);  // buffer = 1000

    // Запускаємо 10,000 агентів як async задачі!
    for id in 0..10_000 {
        let agent_tx = tx.clone();
        tokio::spawn(async move {  // spawn створює легку задачу
            loop {
                // Неблокуюче очікування
                tokio::time::sleep(Duration::from_millis(100)).await;

                // Робота агента...
                let report = DroneReport { drone_id: id, /* ... */ };
```

10,000 задач працюють на ~8 потоках (CPU cores).
Це В 1000 разів ефективніше ніж 10,000 threads!

```rust
                agent_tx.send(report).await;
            }
        });
    }
    drop(tx);

    // Координатор обробляє звіти
    while let Some(report) = rx.recv().await {
        process_report(report);
    }
}
```

```rust
async fn drone_agent(
    mut commands: mpsc::Receiver<Command>,
    shutdown: CancellationToken,
) {
    let mut interval = tokio::time::interval(Duration::from_millis(100));

    loop {
        tokio::select! {
            // Команда від координатора
            Some(cmd) = commands.recv() => {
                handle_command(cmd).await;
            }

            // Тік таймера — час оновити стан
            _ = interval.tick() => {
                sense_and_act().await;
            }

            _ = shutdown.cancelled() => {
                log::info!("Drone shutting down");
                break;
            }
        }
    }
}
```

select! виконує ПЕРШИЙ готовий branch.
Інші branches скасовуються.

Async Channels: Різні типи для різних патернів

Tokio надає кілька типів async каналів:

mpsc (multi-producer, single-consumer)
  Багато агентів → один координатор.
  Найчастіший патерн.

broadcast (single-producer, multi-consumer)
  Один координатор → всі агенти.
  Для подій: "Всім відступати!".

oneshot (one-shot, single value)

mpsc — звіти агентів
broadcast — events для всіх
watch (single value, latest only)
  Для стану, що постійно оновлюється.
  Читач завжди бачить ОСТАННЄ значення.

```rust
use tokio_stream::StreamExt;

/// Обробка телеметрії як Stream
async fn process_telemetry(mut telemetry: impl Stream<Item = Telemetry>) {
    // Stream комбінатори — як для Iterator
    let filtered = telemetry
        .filter(|t| t.battery < 20)  // Тільки низька батарея
        .map(|t| Alert::LowBattery(t.drone_id));

    // Async iteration
    while let Some(alert) = filtered.next().await {
        notify_operator(alert).await;
    }
}

// Aggregation — batch processing
let aggregated = telemetry
    .chunks_timeout(100, Duration::from_secs(1))  // Batch по 100 або 1 сек
```

- Логування події

Streams дозволяють обробляти потік даних реактивно.

Threads vs Async: Коли що використовувати?

Критерій	Threads	Async
Тип задач	CPU-bound	I/O-bound
Масштаб	~1,000	~100,000+
Пам'ять на задачу	1-8 MB	1-10 KB
Parallelism	Справжній	Конкурентність*
Context switch	Дорогий (ОС)	Дешевий
Складність	Простіше	Складніше (futures)
Use case	Обчислення	Очікування

* Tokio multi-threaded runtime дає і parallelism теж!

У MAC зазвичай:

- Async для агентів (багато очікування: команд, таймерів)
- Threads для CPU-heavy задач (pathfinding, ML inference)
- Rayon для data-parallelism (обробка батчів)

Підсумок: Паралелізм для МАС

Лекції 14-21 дали інструменти паралелізму:

Threads (Лекція 14)
- Справжній паралелізм для CPU-bound задач.
- thread::spawn, JoinHandle, move closure.

Channels (Лекції 16-17)
- Message passing замість shared state.
- mpsc: Many Producers → Single Consumer.
- Типобезпечні повідомлення через enum.

async/await (Лекції 18-19)
- Легкі задачі для масштабування до 100,000+.
- Future, .await, кооперативна багатозадачність.

Tokio (Лекції 19-20)
- Production-ready async runtime.
- spawn, select!, async channels, timers.

Streams (Лекція 21)
- Async ітератори для потокових даних.
- Телеметрія, сенсори, aggregation.

→ Частина 4: Архітектури (Actor, ECS, BDI)

Підсумкова лекція

Архітектури МАС

Actor Model • ECS • BDI • Повна інтеграція

Три архітектурні патерни:
• Actor — ізоляція та комунікація
• ECS — масштаб та продуктивність
• BDI — інтелект та автономність

Частина 4 з 4: Архітектури (Лекції 20-24)

О. С. Бичков • 2025

Три архітектури: різні проблеми — різні рішення

Кожна архітектура оптимізована для своєї задачі:

Actor Model (Лекції 20-22)
  Проблема: як організувати комунікацію без спільного стану?
  Рішення: кожен агент — ізольований актор з mailbox.
  Сила: ізоляція, відмовостійкість, distributed systems.

ECS (Лекція 23)
  Проблема: як ефективно обробляти 100,000+ сутностей?
  Рішення: розділити дані (Components) та логіку (Systems).
  Сила: продуктивність, cache-friendly, паралелізм.

BDI (Лекція 24)
  Проблема: як моделювати раціональну поведінку агента?
  Рішення: Beliefs, Desires, Intentions — когнітивна модель.
  Сила: автономність, пояснюваність, адаптивність.

Actor Model: Філософія ізоляції

Actor Model — парадигма, де все є актором.

Кожен актор:
• Має приватний стан (недоступний ззовні)
• Має mailbox (чергу повідомлень)
• Обробляє повідомлення ПОСЛІДОВНО (одне за раз)
• Може створювати інших акторів
• Може надсилати повідомлення

Чому Actor ідеально для MAC?

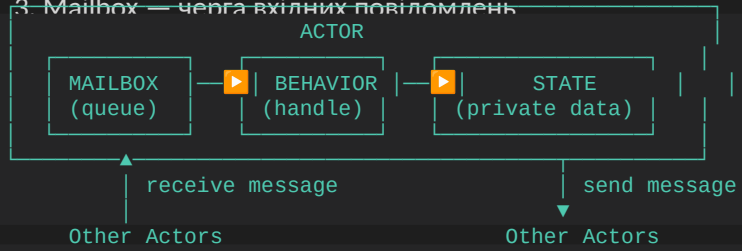
Агент = Актор — природна відповідність!
• Агент має приватний стан (beliefs, battery, position)
• Агент отримує команди через повідомлення
• Агент надсилає звіти координатору
• Агент може "народжувати" субагентів

НЕМАЄ спільного мутабельного стану → НЕМАЄ data races!

Actor Model: Структура актора

Актор має три компоненти:

1. State — приватні дані актора
2. Behavior — логіка обробки повідомлень
3. Mailbox — черга вхідних повідомлень



State ніколи не "виглядає" назовні.
Єдиний спосіб взаємодії — message passing.

```rust
/// Actor trait визначає інтерфейс актора
#[async_trait]
pub trait Actor: Send + 'static {
    type Message: Send;

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

/// DroneActor — агент як актор
pub struct DroneActor {
    id: DroneId,
    position: Position,
    battery: u8,
    beliefs: BeliefBase,
}

impl Actor for DroneActor {
```

handle() — серце актора. Отримує повідомлення, оновлює стан.

```rust
    async fn handle(&mut self, msg: DroneMessage, ctx: &mut ActorContext<Self>) {
        match msg {
            DroneMessage::MoveTo(target) => {
                self.position = self.calculate_path(target);
            }
            DroneMessage::GetStatus { reply } => {
                // reply — oneshot канал для відповіді
                let _ = reply.send(self.status());
            }
            DroneMessage::Shutdown => ctx.stop(),
        }
```

```rust
/// Handle для взаємодії з DroneActor
#[derive(Clone)]  // Можна передавати між потоками
pub struct DroneHandle {
    id: DroneId,
    sender: mpsc::Sender<DroneMessage>,
}

impl DroneHandle {
    /// Fire-and-forget: надіслати команду без очікування
    pub async fn move_to(&self, target: Position) -> Result<(), SendError> {
        self.sender.send(DroneMessage::MoveTo(target)).await
    }

    /// Request-Reply: надіслати запит і чекати відповідь
    pub async fn get_status(&self) -> Result<DroneStatus, Error> {
        let (tx, rx) = oneshot::channel();  // Канал для відповіді
        self.sender.send(DroneMessage::GetStatus { reply: tx }).await?;
        rx.await.map_err(|_| Error::ActorDead)
    }
}

// Використання — простий API
let drone = spawn_drone_actor(config);
drone.move_to(target).await?;  // Команда
let status = drone.get_status().await?;  // Запит
```

ECS: Філософія Data-Oriented Design

ECS (Entity-Component-System) — архітектура для МАСШТАБУ.

Традиційний OOP:
```
class Drone {
  position, velocity, battery, camera, weapon...
  move(), scan(), fire()...
}
```
Проблема: всі дані розкидані по пам'яті → cache misses!

ECS перевертає дизайн:

- Entity — просто число (ID), без даних і логіки
- Component — ТІЛЬКИ дані, без логіки
- System — ТІЛЬКИ логіка, без даних

Дані одного типу зберігаються РАЗОМ:
[Position1, Position2, Position3, ...] — один масив
[Battery1, Battery2, Battery3, ...] — інший масив

Це cache-friendly → швидка обробка 100,000+ сутностей!

```rust
// System руху — обробляє ВСІ entities з Position та Velocity
fn movement_system(
    time: Res<Time>,
    mut query: Query<(&mut Position, &Velocity)>,  // Запит компонентів
) {
    let dt = time.delta_seconds();
    for (mut pos, vel) in query.iter_mut() {
        pos.0 += vel.0 * dt;  // Оновлюємо позицію
    }
}

// System тільки для Scout дронів
fn scout_scan_system(
    scouts: Query<(&Position, &Camera), With<Scout>>,  // Тільки Scout!
    targets: Query<&Position, With<Target>>,
) { /* ... */ }
```

Query<...> — декларативний запит: "дай всі entities з цими components"

ECS: Чому це швидко?

1. Cache-friendly memory layout

OOP (Array of Structs):
[Drone1{pos,vel,bat}, Drone2{pos,vel,bat}, ...]
Обробка тільки position → завантажуємо ВСЕ → cache waste!

ECS (Struct of Arrays):
positions: [pos1, pos2, pos3, ...] — послідовно в пам'яті
velocities: [vel1, vel2, vel3, ...]
Обробка positions → тільки positions в cache → ефективно!

2. Паралелізм з коробки

Systems що не конфліктують (різні components) — паралельні!
movement_system (Position, Velocity)
battery_system (Battery)
→ Можуть працювати одночасно!

3. Гнучка композиція

Додати камеру дрону? → Просто insert(Camera {...}).
Не треба міняти клас чи ієрархію.

BDI: Філософія раціональних агентів

BDI (Beliefs-Desires-Intentions) — когнітивна архітектура.

Ідея: моделювати агента як раціональну ОСОБИСТІСТЬ,
що має знання, цілі та плани — як людина.

Beliefs (Переконання) — що агент ЗНАЄ про світ:
- Моя позиція (10, 20, 50)
- Батарея 45%
- Ворог на позиції (100, 80)
- Союзник #7 летить на базу

Desires (Бажання) — чого агент ХОЧЕ:
- Знищити ворога
- Повернутись на базу
- Зберегти енергію
- Уникнути зіткнення

Intentions (Наміри) — до чого агент ЗОБОВ'ЯЗАВСЯ:
- Виконую патрулювання (план: waypoints 1→2→3→base)

BDI: Цикл обробки агента

BDI агент працює в циклі обдумування:

loop {
1. Perception — отримати дані від сенсорів
2. Belief Revision — оновити beliefs на основі нових даних
3. Option Generation — визначити можливі desires
4. Deliberation — обрати intentions з desires
5. Means-End Reasoning — знайти план для intention
6. Execute — виконати наступну дію плану

}

Ключові питання deliberation:
• Які desires досяжні з поточних beliefs?
• Які desires мають вищий пріоритет?
• Чи сумісні нові desires з поточними intentions?

Commitment: агент НЕ кидає intention при першій зміні!
Single-minded: виконує до успіху або неможливості.

```rust
pub enum DesireCategory {
    Safety = 4,      // Найвищий: уникнути катастрофи
    Survival = 3,    // Зберегти агента
    Mission = 2,     // Виконати завдання
    Efficiency = 1,  // Оптимізація
}

// Приклади desires за категоріями:
// Safety (пріоритет ~250):
//   AvoidCollision, EvadeHostileFire

// Survival (пріоритет ~200):
//   ReturnToBase (при battery < 20%), MaintainComms

// Mission (пріоритет ~100):
//   CompletePatrol, DestroyTarget, ProvideSupport

// Efficiency (пріоритет ~50):
//   OptimizePath, ConserveEnergy

// Правило: Safety ЗАВЖДИ переважає Mission!
// Навіть якщо ціль перед носом — якщо батарея 5%, летимо на базу.
```

Це формалізація: "Безпека важливіша за місію."

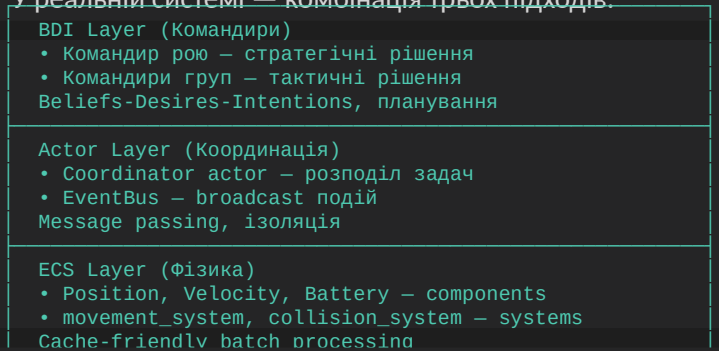
Порівняння трьох архітектур

Критерій	Actor	ECS	BDI
Масштаб	~10K акторів	~1M сутностей	~100-1K агентів
Фокус	Ізоляція, комунікація	Продуктивність	Інтелект
Стан	Приватний в акторі	Спільний (Components)	BeliefBase
Логіка	handle() method	Systems	Deliberation
Комунікація	Messages	Events, Resources	Beliefs update
Parallelism	Per-actor	Per-system	Per-agent
Пояснюваність	Середня	Низька	Висока
Складність	Середня	Низька	Висока

Немає "найкращої" архітектури — є ПРАВИЛЬНА для задачі.
У реальних MAC часто комбінують!

Гібридна архітектура рою БПЛА

У реальній системі — комбінація трьох підходів:



ECS — 1000 дронів з ефективною фізикою
Actor — координація та комунікація
BDI — інтелектуальні рішення командирів

```rust
pub struct SwarmSystem {
    // ECS World — фізика та стан всіх дронів
    world: bevy_ecs::World,
    schedule: bevy_ecs::Schedule,

    // Actor handles — комунікація
    coordinator: CoordinatorHandle,
    drones: HashMap<DroneId, DroneHandle>,

    // BDI commanders — інтелект
    swarm_commander: BDIAgent,
    group_commanders: HashMap<GroupId, BDIAgent>,

    // Async infrastructure
    event_bus: broadcast::Sender<SwarmEvent>,
    telemetry: mpsc::Receiver<Telemetry>,
    shutdown: CancellationToken,
}

impl SwarmSystem {
    pub async fn tick(&mut self) {
        // 1. ECS: оновити фізику всіх дронів
        self.schedule.run(&mut self.world);

        // 2. Actor: обробити повідомлення
        self.coordinator.process_messages().await;

        // 3. BDI: командири приймають рішення
        for commander in self.group_commanders.values_mut() {
            let actions = commander.tick(perception);
```

Всі 20 лекцій разом

Лекції	Технології	Роль у системі
4-5	Vec, HashMap	Дані агентів, BeliefBase
6-7	Option, Result	Обробка невизначеності
8-9	Traits, Generics	Абстракції Agent, Registry<A>
10-11	Box, Rc/Arc	Рекурсивні плани, спільні дані
12-13	RefCell, Lifetimes	Interior mutability, views
14-15	Threads, Arc<Mutex>	Паралельні агенти
16-17	Channels	Message passing
18-19	async/await, Tokio	Масштабування до 10,000+
20-21	Async channels, Streams	Комунікація, телеметрія
22	Actor Model	Ізоляція, координація
23	ECS	Фізика, масові симуляції
24	BDI	Інтелектуальні командири

Фінальний підсумок курсу

20 лекцій → Production-ready мультиагентна система:

✅ Структури даних (4-9)
Vec, HashMap, Option, Result, Traits, Generics
→ Моделювання даних та абстракцій агента

✅ Керування пам'яттю (10-13)
Box, Rc/Arc, RefCell, Lifetimes
→ Складні структури, спільні дані, безпечні посилання

✅ Паралелізм (14-21)
Threads, Channels, async/await, Tokio, Streams
→ Від 100 до 100,000 агентів

✅ Архітектури (20-24)
Actor Model, ECS, BDI
→ Ізоляція + Масштаб + Інтелект

🤖 Результат: Рій БПЛА з гібридною архітектурою,
що масштабується, надійний та інтелектуальний.

Rust + MAC = Production-Ready!

20 технологій → 3 архітектури → 1 система

Структури даних • Smart Pointers • Паралелізм
Async • Channels • Streams • Actor • ECS • BDI

Питання?