

Лекція 15

Практикум: Локальний рій агентів

Інтеграція threads, Arc, Mutex, channels



Повноцінна мультиагентна система на одній машині

Компоненти системи:

- Агенти-БПЛА (потоки)
- Координатор
- Спільна карта світу
- Комунікація через канали
- Синхронізація станів

Частина 1: Архітектура та базові компоненти

План практикуму

Частина 1: Архітектура

1. Огляд системи
2. Архітектура рою
3. Типи даних
4. Position та Area
5. AgentState enum
6. Agent структура
7. Message types
8. Command types
9. WorldMap структура
10. SharedState

Частина 2: Реалізація

11. Agent implementation
12. Agent::run() loop
13. Coordinator структура
14. Coordinator::run()
15. Spawning agents
16. Main function
17. Graceful shutdown
18. Testing
19. Розширення
20. Повний код



Огляд системи: Локальний рій БПЛА

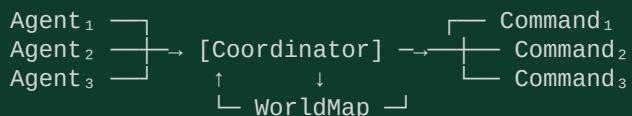
Мета: Створити симуляцію рою БПЛА на одній машині

Компоненти:

- 5-10 агентів-БПЛА (кожен у своєму потоці)
- 1 координатор (окремий потік)
- Спільна карта світу (`Arc<RwLock<WorldMap>>`)
- Канали для комунікації

Поведінка агентів:

- Патрулювання призначененої зони
- Виявлення підлітків



Архітектура системи

Agents (Threads)

Agent 1
├ position
├ state
└ battery
└ patrol_area

Agent 2
├ ...

Agent N
├ ...

Coordinator (Thread)

- Отримує повідомлення
- Оновлює WorldMap
- Приймає рішення
- Надсилає команди

Channels:

- from_agents: Receiver
- to_agents: Vec<Sender>

Shared:

- Arc<RwLock<WorldMap>>

Shared State

WorldMap:
└ cells: HashMap
└ targets: Vec
└ explored: HashSet

Stats:
└ messages: AtomicU64
└ moves: AtomicU64
└ detections: AtomicU64

Flags:
└ shutdown: AtomicBool

```
use std::ops::{Add, Sub};

/// Позиція на 2D карті
#[derive(Debug, Clone, Copy, PartialEq)]
pub struct Position {
    pub x: f64,
    pub y: f64,
}

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

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

    /// Напрямок до іншої позиції (радіани)
    pub fn direction_to(&self, other: &Position) -> f64 {
        let dx = other.x - self.x;
        let dy = other.y - self.y;
        dy.atan2(dx)
    }

    /// Рух у напрямку на відстань
    pub fn move_towards(&self, target: &Position, distance: f64) -> Position {
```

```
impl Add for Position {
    type Output = Position;
    fn add(self, other: Position) -> Position {
        Position::new(self.x + other.x, self.y + other.y)
    }
}

impl Sub for Position {
    type Output = Position;
    fn sub(self, other: Position) -> Position {
        Position::new(self.x - other.x, self.y - other.y)
    }
}

// Hash для використання в HashMap
impl std::hash::Hash for Position {
    fn hash<H: std::hash::Hasher>(&self, state: &mut H) {
        // Дискретизуємо для hash
        (self.x as i64).hash(state);
        (self.y as i64).hash(state);
    }
}

impl Eq for Position {} // Потрібно для HashMap key

// Display для зручного виводу
impl std::fmt::Display for Position {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        write!(f, "({:.1}, {:.1})", self.x, self.y)
    }
}
```

```
// Прямоугольная область
#[derive(Debug, Clone, Copy)]
pub struct Area {
    pub min: Position,
    pub max: Position,
}

impl Area {
    pub fn new(x1: f64, y1: f64, x2: f64, y2: f64) -> Self {
        Area {
            min: Position::new(x1.min(x2), y1.min(y2)),
            max: Position::new(x1.max(x2), y1.max(y2)),
        }
    }

    /// Чи містить позицію
    pub fn contains(&self, pos: &Position) -> bool {
        pos.x >= self.min.x && pos.x <= self.max.x &&
        pos.y >= self.min.y && pos.y <= self.max.y
    }

    /// Центр області
    pub fn center(&self) -> Position {
        Position::new(
            (self.min.x + self.max.x) / 2.0,
            (self.min.y + self.max.y) / 2.0,
        )
    }

    /// Випадкова позиція в області
}
```

```
// Стан агента – машина станів
#[derive(Debug, Clone, PartialEq)]
pub enum AgentState {
    /// Очікування команд
    Idle,

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

    /// Рух до цілі
    MovingTo {
        target: Position,
        reason: MoveReason,
    },

    /// Сканування території
    Scanning {
        center: Position,
        radius: f64,
    },

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

    /// Аварійний стан
}
```

```
// Ідентифікатор цілі
#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash)]
pub struct TargetId(pub u32);

// Тип цілі
#[derive(Debug, Clone, PartialEq)]
pub enum TargetType {
    Unknown,
    Friendly,
    Neutral,
    Hostile,
}

// Виявлення ціль
#[derive(Debug, Clone)]
pub struct Target {
    pub id: TargetId,
    pub target_type: TargetType,
    pub position: Position,
    pub velocity: Option<Position>, // Якщо рухається
    pub confidence: f64,           // 0.0 - 1.0
    pub last_seen: std::time::Instant,
}

// Інформація про виявлення
#[derive(Debug, Clone)]
pub struct Detection {
    pub target: Target,
    pub detected_by: AgentId,
    pub timestamp: std::time::Instant,
```

```
// Унікальний ідентифікатор агента
#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash)]
pub struct AgentId(pub u32);

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

/// Інформація про агента для координатора
#[derive(Debug, Clone)]
pub struct AgentInfo {
    pub id: AgentId,
    pub position: Position,
    pub state: AgentState,
    pub battery: u8,
    pub last_update: std::time::Instant,
}

impl AgentInfo {
    pub fn new(id: AgentId) -> Self {
        AgentInfo {
            id,
            position: Position::new(0.0, 0.0),
            state: AgentState::Idle,
            battery: 100,
            last_update: std::time::Instant::now(),
        }
    }
}
```

```
use std::time::Instant;

/// Повідомлення від агента до координатора
#[derive(Debug, Clone)]
pub enum AgentMessage {
    /// Оновлення позиції
    PositionUpdate {
        agent_id: AgentId,
        position: Position,
        battery: u8,
    },
    /// Зміна стану
    StateChanged {
        agent_id: AgentId,
        new_state: AgentState,
    },
    /// Виявлено ціль
    TargetDetected {
        agent_id: AgentId,
        target: Target,
    },
    /// Втрачено ціль з поля зору
    TargetLost {
        agent_id: AgentId,
        target_id: TargetId,
    },
}
```

```
// Команда від координатора до агента
#[derive(Debug, Clone)]
pub enum Command {
    // Рухатись до позиції
    MoveTo {
        target: Position,
    },
    // Патрулювати зону
    Patrol {
        area: Area,
    },
    // Сканувати область
    Scan {
        center: Position,
        radius: f64,
    },
    // Переслідувати ціль
    PursueTarget {
        target_id: TargetId,
        last_known_position: Position,
    },
    // Повернутись на базу
    ReturnToBase,
    // Зупинитись
    Halt,
```

```
use std::collections::{HashMap, HashSet};

/// Тип клітинки карти
#[derive(Debug, Clone, PartialEq)]
pub enum CellType {
    Unknown,
    Empty,
    Obstacle,
    Target(TargetId),
}

/// Карта світу (shared state)
#[derive(Debug)]
pub struct WorldMap {
    /// Клітинки карти (дискретизовані координати)
    pub cells: HashMap<(i32, i32), CellType>,

    /// Відомі цілі
    pub targets: HashMap<TargetId, Target>,

    /// Досліджені позиції
    pub explored: HashSet<(i32, i32)>,

    /// Розміри карти
    pub bounds: Area,
}

impl WorldMap {
    pub fn new(bounds: Area) -> Self {
        WorldMap {

```

```
impl WorldMap {
    /// Дискретизація позиції до клітинки
    fn pos_to_cell(pos: &Position) -> (i32, i32) {
        (pos.x.floor() as i32, pos.y.floor() as i32)
    }

    /// Позначити як досліджене
    pub fn mark_explored(&mut self, pos: &Position) {
        let cell = Self::pos_to_cell(pos);
        self.explored.insert(cell);
        self.cells.entry(cell).or_insert(CellType::Empty);
    }

    /// Додати або оновити ціль
    pub fn update_target(&mut self, target: Target) {
        let cell = Self::pos_to_cell(&target.position);
        self.cells.insert(cell, CellType::Target(target.id));
        self.targets.insert(target.id, target);
    }

    /// Видалити ціль
    pub fn remove_target(&mut self, target_id: TargetId) {
        if let Some(target) = self.targets.remove(&target_id) {
            let cell = Self::pos_to_cell(&target.position);
            self.cells.insert(cell, CellType::Empty);
        }
    }

    /// Відсоток дослідження
    pub fn exploration_percent(&self) -> f64 {
```

```
use std::sync::{Arc, RwLock, atomic::{AtomicBool, AtomicU64, Ordering}};

/// Статистика рою (lock-free)
pub struct SwarmStats {
    pub total_messages: AtomicU64,
    pub total_moves: AtomicU64,
    pub total_detections: AtomicU64,
    pub active_agents: AtomicU64,
}

impl SwarmStats {
    pub fn new() -> Self {
        SwarmStats {
            total_messages: AtomicU64::new(0),
            total_moves: AtomicU64::new(0),
            total_detections: AtomicU64::new(0),
            active_agents: AtomicU64::new(0),
        }
    }
}

/// Спільний стан системи
pub struct SharedState {
    pub world_map: Arc<RwLock<WorldMap>>,
    pub stats: Arc<SwarmStats>,
    pub shutdown: Arc<AtomicBool>,
}

impl SharedState {
    pub fn new(bounds: Area) -> Self {
```

```
use std::sync::mpsc::{Sender, Receiver};

/// Агент БПЛА
pub struct Agent {
    // Ідентифікація
    pub id: AgentId,

    // Стан
    pub position: Position,
    pub state: AgentState,
    pub battery: u8,

    // Конфігурація
    pub base_position: Position,
    pub patrol_area: Option<Area>,
    pub speed: f64,
    pub scan_radius: f64,

    // Комунікація
    pub to_coordinator: Sender<AgentMessage>,
    pub from_coordinator: Receiver<Command>,

    // Спільний стан
    pub shared: SharedState,

    // Внутрішній стан
    waypoints: Vec<Position>,
    current_waypoint: usize,
}
```

```
impl Agent {
    pub fn new(
        id: AgentId,
        base_position: Position,
        to_coordinator: Sender<AgentMessage>,
        from_coordinator: Receiver<Command>,
        shared: SharedState,
    ) -> Self {
        Agent {
            id,
            position: base_position,
            state: AgentState::Idle,
            battery: 100,
            base_position,
            patrol_area: None,
            speed: 1.0,
            scan_radius: 5.0,
            to_coordinator,
            from_coordinator,
            shared,
            waypoints: Vec::new(),
            current_waypoint: 0,
        }
    }

    /// Перевірка чи потрібно повернутись на базу
    fn should_return_to_base(&self) -> bool {
        self.battery <= Self::LOW_BATTERY_THRESHOLD
    }
}
```

```
impl Agent {
    /// Рух на один крок до цілі
    fn move_towards(&mut self, target: &Position) {
        if self.position.distance_to(target) > self.speed {
            self.position = self.position.move_towards(target, self.speed);
        } else {
            self.position = *target;
        }

        // Витрата батареї
        self.battery = self.battery.saturating_sub(Self::BATTERY_DRAIN_PER_TICK);

        // Оновлення статистики
        self.shared.stats.total_moves.fetch_add(1, Ordering::Relaxed);

        // Позначити як досліджене
        if let Ok(mut map) = self.shared.world_map.write() {
            map.mark_explored(&self.position);
        }
    }

    /// Перевірка досягнення цілі
    fn reached_target(&self, target: &Position) -> bool {
        self.position.distance_to(target) < 0.5
    }

    /// Генерація waypoints для патрулювання
    fn generate_patrol_waypoints(&mut self, area: &Area) {
        self.waypoints = vec![
            Position::new(area.min.x, area.min.y),
```

```
impl Agent {
    /// Сканування області навколо (симуляція)
    fn scan_area(&self) -> Vec<Target> {
        use rand::Rng;
        let mut rng = rand::thread_rng();
        let mut detected = Vec::new();

        // Симуляція: 10% шанс виявити ціль
        if rng.gen_bool(0.1) {
            let target = Target {
                id: TargetId(rng.gen()),
                target_type: TargetType::Unknown,
                position: Position::new(
                    self.position.x + rng.gen_range(-self.scan_radius..self.scan_radius),
                    self.position.y + rng.gen_range(-self.scan_radius..self.scan_radius),
                ),
                velocity: None,
                confidence: rng.gen_range(0.5..1.0),
                last_seen: std::time::Instant::now(),
            };
            detected.push(target);
        }
        detected
    }

    /// Надіслати повідомлення координатору
    fn send_message(&self, msg: AgentMessage) {
        if self.to_coordinator.send(msg).is_ok() {
            self.shared.stats.total_messages.fetch_add(1, Ordering::Relaxed);
        }
    }
}
```

```
impl Agent {
    /// Обробка команди від координатора
    fn handle_command(&mut self, cmd: Command) {
        println!("[{} Received command: {:?}", self.id, cmd);

        match cmd {
            Command::MoveTo { target } => {
                self.state = AgentState::MovingTo {
                    target,
                    reason: MoveReason::Command,
                };
            }

            Command::Patrol { area } => {
                self.generate_patrol_waypoints(&area);
                self.patrol_area = Some(area);
                self.state = AgentState::Patrolling {
                    area,
                    waypoint_index: 0,
                };
            }

            Command::ReturnToBase => {
                self.state = AgentState::Returning {
                    base: self.base_position,
                };
            }

            Command::Halt => {
                self.state = AgentState::Idle;
            }
        }
    }
}
```

```
impl Agent {  
    /// Один крок симуляції  
    fn tick(&mut self) {  
        // Перевірка батареї  
        if self.should_return_to_base() && !matches!(self.state, AgentState::Returning { .. }) {  
            self.send_message(AgentMessage::LowBattery {  
                agent_id: self.id,  
                level: self.battery,  
            });  
            self.state = AgentState::Returning { base: self.base_position };  
        }  
    }  
}
```

// Дія залежно від стану

```
match &self.state.clone() {  
    AgentState::Idle => {  
        // Нічого не робимо  
    }  
}
```

```
AgentState::Patrolling { area, waypoint_index } => {  
    self.tick_patrol(*waypoint_index);  
}  
}
```

```
AgentState::MovingTo { target, .. } => {  
    self.move_towards(target);  
    if self.reached_target(target) {  
        self.state = AgentState::Idle;  
    }  
}  
}
```

```
AgentState::Returning { base } => {
```

```
impl Agent {
    fn tick_patrol(&mut self, waypoint_index: usize) {
        if self.waypoints.is_empty() {
            return;
        }

        let target = self.waypoints[waypoint_index];
        self.move_towards(&target);

        // Сканування під час руху
        let detected = self.scan_area();
        for target in detected {
            self.send_message(AgentMessage::TargetDetected {
                agent_id: self.id,
                target: target.clone(),
            });
        }

        // Оновлюємо карту
        if let Ok(mut map) = self.shared.world_map.write() {
            map.update_target(target);
        }

        self.shared.stats.total_detections.fetch_add(1, Ordering::Relaxed);
    }

    // Досягли waypoint – переходимо до наступного
    if self.reached_target(&target) {
        let next_index = (waypoint_index + 1) % self.waypoints.len();
        if let Some(area) = self.patrol_area {
            self.state = AgentState::Patrolling {
```

Підсумок: Частина 1

Визначено базові типи:

- Position, Area — геометрія
- AgentState — машина станів
- Target, Detection — цілі
- AgentId, AgentInfo — ідентифікація

Повідомлення та команди:

- AgentMessage — від агента до координатора
- Command — від координатора до агента

Спільний стан:

- WorldMap — карта світу (RwLock)
- SwarmStats — статистика (Atomic)
- SharedState — обгортка

→ Частина 2: Coordinator, main(), повний код
Agent структура:

- Поля стану та конфігурації
- Методи руху, сканування
- handle_command, tick

Лекція 15 (продовження)

Coordinator та запуск системи

Повна реалізація локального рою

Coordinator • main() • graceful shutdown • testing



Запуск рою: координація, моніторинг, зупинка

Частина 2: Coordinator та main()

План (Частина 2)

- | | |
|--|--|
| <ul style="list-style-type: none">1. Agent::run() — головний цикл2. Coordinator структура3. Coordinator::new()4. Coordinator::handle_message()5. Coordinator::assign_patrol_areas()6. Coordinator::run()7. Coordinator прийняття рішень8. SwarmManager — менеджер рою | <ul style="list-style-type: none">9. Spawning agents10. main() — точка входу11. Graceful shutdown12. Обробка Ctrl+C13. Моніторинг та логування14. Тестування15. Можливі розширення16. Повна структура проекту |
|--|--|

```
impl Agent {
    /// Головний цикл агента (виконується в окремому потоці)
    pub fn run(mut self) {
        println!("[{}]\nStarted at {}", self.id, self.position);

        // Збільшуємо лічильник активних агентів
        self.shared.stats.active_agents.fetch_add(1, Ordering::SeqCst);

        // Надсилаємо початкову позицію
        self.send_position_update();

        loop {
            // Перевірка shutdown
            if self.shared.shutdown.load(Ordering::SeqCst) {
                println!("[{}]\nShutdown signal received", self.id);
                break;
            }

            // Обробка команд (non-blocking)
            while let Ok(cmd) = self.from_coordinator.try_recv() {
                if matches!(cmd, Command::Shutdown) {
                    break;
                }
                self.handle_command(cmd);
            }

            // Крок симуляції
            self.tick();

            // Надсилаємо оновлення позиції
        }
    }
}
```

```
impl Agent {
    /// Надіслати оновлення позиції
    fn send_position_update(&self) {
        self.send_message(AgentMessage::PositionUpdate {
            agent_id: self.id,
            position: self.position,
            battery: self.battery,
        });
    }

    /// Надіслати зміну стану
    fn send_state_change(&self) {
        self.send_message(AgentMessage::StateChanged {
            agent_id: self.id,
            new_state: self.state.clone(),
        });
    }

    /// Отримати інформацію для координатора
    pub fn info(&self) -> AgentInfo {
        AgentInfo {
            id: self.id,
            position: self.position,
            state: self.state.clone(),
            battery: self.battery,
            last_update: std::time::Instant::now(),
        }
    }
}
```

```
use std::sync::mpsc::{Sender, Receiver};
use std::collections::HashMap;

/// Координатор рою
pub struct Coordinator {
    /// Канал для отримання повідомлень від агентів
    from_agents: Receiver<AgentMessage>,

    /// Канали для надсилання команд агентам
    to_agents: HashMap<AgentId, Sender<Command>>,

    /// Інформація про агентів
    agents: HashMap<AgentId, AgentInfo>,

    /// Спільний стан
    shared: SharedState,

    /// Конфігурація
    config: CoordinatorConfig,
}

/// Конфігурація координатора
pub struct CoordinatorConfig {
    pub patrol_area: Area,
    pub low_battery_threshold: u8,
    pub max_targets_per_agent: usize,
}

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

```
impl Coordinator {
    pub fn new(
        from_agents: Receiver<AgentMessage>,
        shared: SharedState,
        config: CoordinatorConfig,
    ) -> Self {
        Coordinator {
            from_agents,
            to_agents: HashMap::new(),
            agents: HashMap::new(),
            shared,
            config,
        }
    }

    /// Зареєструвати агента
    pub fn register_agent(&mut self, id: AgentId, command_tx: Sender<Command>) {
        self.to_agents.insert(id, command_tx);
        self.agents.insert(id, AgentInfo::new(id));
        println!("[Coordinator] Registered {}", id);
    }

    /// Надіслати команду агенту
    pub fn send_command(&self, agent_id: AgentId, cmd: Command) {
        if let Some(tx) = self.to_agents.get(&agent_id) {
            if tx.send(cmd).is_err() {
                println!("[Coordinator] Failed to send command to {}", agent_id);
            }
        }
    }
}
```

```
impl Coordinator {
    /// Обробка повідомлення від агента
    fn handle_message(&mut self, msg: AgentMessage) {
        match msg {
            AgentMessage::PositionUpdate { agent_id, position, battery } => {
                if let Some(info) = self.agents.get_mut(&agent_id) {
                    info.position = position;
                    info.battery = battery;
                    info.last_update = std::time::Instant::now();
                }
            }

            AgentMessage::StateChanged { agent_id, new_state } => {
                if let Some(info) = self.agents.get_mut(&agent_id) {
                    info.state = new_state;
                }
            }

            AgentMessage::TargetDetected { agent_id, target } => {
                println!("[Coordinator] {} detected target {:?} at {}", agent_id, target.id, target.position);

                // Оновлюємо карту
                if let Ok(mut map) = self.shared.world_map.write() {
                    map.update_target(target.clone());
                }

                // Можна призначити інших агентів для підтримки
                self.handle_target_detected(agent_id, target);
            }
        }
    }
}
```

```
impl Coordinator {
    /// Реакція на виявлення цілі
    fn handle_target_detected(&mut self, detector_id: AgentId, target: Target) {
        // Знайти найближчих агентів для підтримки
        let mut available: Vec<_> = self.agents.iter()
            .filter(|(id, info)| {
                **id != detector_id &&
                matches!(info.state, AgentState::Patrolling { .. } | AgentState::Idle) &&
                info.battery > self.config.low_battery_threshold
            })
            .map(|(id, info)| (*id, info.position.distance_to(&target.position)))
            .collect();

        // Сортуємо по відстані
        available.sort_by(|a, b| a.1.partial_cmp(&b.1).unwrap());

        // Призначаємо найближчого для підтримки
        if let Some((support_id, _)) = available.first() {
            println!("[Coordinator] Assigning {} to support {} at target {:?}", support_id, detector_id, target.id);

            self.send_command(*support_id, Command::PursueTarget {
                target_id: target.id,
                last_known_position: target.position,
            });
        }
    }
}
```

```
impl Coordinator {
    /// Розподіл патрульних зон між агентами
    pub fn assign_patrol_areas(&self) {
        let agent_ids: Vec<_> = self.agents.keys().copied().collect();
        let n = agent_ids.len();

        if n == 0 { return; }

        // Ділимо область на вертикальні смуги
        let area = &self.config.patrol_area;
        let width = (area.max.x - area.min.x) / n as f64;

        for (i, agent_id) in agent_ids.iter().enumerate() {
            let patrol_area = Area::new(
                area.min.x + width * i as f64,
                area.min.y,
                area.min.x + width * (i + 1) as f64,
                area.max.y,
            );

            println!("[Coordinator] Assigning {} to patrol {:?}", agent_id, patrol_area);
            self.send_command(*agent_id, Command::Patrol { area: patrol_area });
        }
    }

    /// Перерозподіл зон (якщо агент вибув)
    pub fn reassign_patrol_areas(&self) {
        // Можна викликати коли агент відключився
        self.assign_patrol_areas();
    }
}
```

```
impl Coordinator {
    /// Головний цикл координатора
    pub fn run(mut self) {
        println!("[Coordinator] Started");

        // Початкове призначення зон
        std::thread::sleep(std::time::Duration::from_millis(500));
        self.assign_patrol_areas();

        loop {
            // Перевірка shutdown
            if self.shared.shutdown.load(Ordering::SeqCst) {
                println!("[Coordinator] Shutdown signal received");
                self.broadcast(Command::Shutdown);
                break;
            }

            // Обробка повідомлень з timeout
            match self.from_agents.recv_timeout(Duration::from_millis(100)) {
                Ok(msg) => self.handle_message(msg),
                Err(mspsc::RecvTimeoutError::Timeout) => {
                    // Періодичні задачі
                    self.periodic_tasks();
                }
                Err(mspsc::RecvTimeoutError::Disconnected) => {
                    println!("[Coordinator] All agents disconnected");
                    break;
                }
            }
        }
    }
}
```

```
impl Coordinator {
    /// Периодичні задачі координатора
    fn periodic_tasks(&mut self) {
        // Перевірка "мертвих" агентів
        let now = std::time::Instant::now();
        let timeout = std::time::Duration::from_secs(5);

        let dead_agents: Vec<_> = self.agents.iter()
            .filter(|(_ , info)| now.duration_since(info.last_update) > timeout)
            .map(|(id, _)| *id)
            .collect();

        for agent_id in dead_agents {
            println!("[Coordinator] Agent {} seems dead, removing", agent_id);
            self.agents.remove(&agent_id);
            self.to_agents.remove(&agent_id);
        }

        // Вивід статистики (кожні N секунд)
        static LAST_STATS: std::sync::atomic::AtomicU64 = std::sync::atomic::AtomicU64::new(0);
        let now_ms = now.elapsed().as_millis() as u64;
        if now_ms - LAST_STATS.load(Ordering::Relaxed) > 5000 {
            self.print_stats();
            LAST_STATS.store(now_ms, Ordering::Relaxed);
        }
    }

    fn print_stats(&self) {
        let stats = &self.shared.stats;
        println!("[Stats] Messages: {}, Moves: {}, Detections: {}, Active: {}",

```

```
use std::thread::JoinHandle;

/// Менеджер рою – керує всіма компонентами
pub struct SwarmManager {
    shared: SharedState,
    agent_handles: Vec<JoinHandle<()>>,
    coordinator_handle: Option<JoinHandle<()>>,
    coordinator_tx: Sender<AgentMessage>,
}

impl SwarmManager {
    pub fn new(bounds: Area) -> Self {
        let shared = SharedState::new(bounds);
        let (coordinator_tx, coordinator_rx) = mpsc::channel();

        // Створюємо координатора
        let coord_shared = SharedState {
            world_map: Arc::clone(&shared.world_map),
            stats: Arc::clone(&shared.stats),
            shutdown: Arc::clone(&shared.shutdown),
        };

        let config = CoordinatorConfig {
            patrol_area: bounds,
            ..Default::default()
        };

        let mut coordinator = Coordinator::new(coordinator_rx, coord_shared, config);

        SwarmManager {
            shared,
            agent_handles: Vec::new(),
            coordinator_handle: Some(coordinator_handle),
            coordinator_tx,
        }
    }
}
```

```
impl SwarmManager {
    /// Створити та запустити агента
    pub fn spawn_agent(&mut self, id: u32, base: Position) -> AgentId {
        let agent_id = AgentId(id);

        // Канали для агента
        let (cmd_tx, cmd_rx) = mpsc::channel();

        // Спільний стан для агента
        let agent_shared = SharedState {
            world_map: Arc::clone(&self.shared.world_map),
            stats: Arc::clone(&self.shared.stats),
            shutdown: Arc::clone(&self.shared.shutdown),
        };

        // Створюємо агента
        let agent = Agent::new(
            agent_id,
            base,
            self.coordinator_tx.clone(),
            cmd_rx,
            agent_shared,
        );

        // Запускаємо в окремому потоці
        let handle = std::thread::Builder::new()
            .name(format!("agent-{}", id))
            .spawn(move || agent.run())
            .expect("Failed to spawn agent thread");
    }
}
```

```
impl SwarmManager {  
    /// Запустити координатора  
    pub fn start_coordinator(&mut self, mut coordinator: Coordinator) {  
        let handle = std::thread::Builder::new()  
            .name("coordinator".to_string())  
            .spawn(move || coordinator.run())  
            .expect("Failed to spawn coordinator thread");  
  
        self.coordinator_handle = Some(handle);  
        println!("[SwarmManager] Coordinator started");  
    }  
  
    /// Graceful shutdown  
    pub fn shutdown(&mut self) {  
        println!("[SwarmManager] Initiating shutdown...");  
  
        // Встановлюємо shutdown flag  
        self.shared.shutdown.store(true, Ordering::SeqCst);  
  
        // Чекаємо на координатора  
        if let Some(handle) = self.coordinator_handle.take() {  
            let _ = handle.join();  
            println!("[SwarmManager] Coordinator stopped");  
        }  
  
        // Чекаємо на агентів  
        for handle in self.agent_handles.drain(..) {  
            let _ = handle.join();  
        }  
    }  
}
```

```
use std::time::Duration;

fn main() {
    println!("== Swarm Simulation ==");

    // Конфігурація
    let world_bounds = Area::new(0.0, 0.0, 100.0, 100.0);
    let num_agents = 5;

    // Створюємо менеджер
    let mut manager = SwarmManager::new(world_bounds);

    // Створюємо агентів
    let base_positions = vec![
        Position::new(10.0, 10.0),
        Position::new(90.0, 10.0),
        Position::new(50.0, 50.0),
        Position::new(10.0, 90.0),
        Position::new(90.0, 90.0),
    ];
}

for (i, base) in base_positions.into_iter().enumerate() {
    manager.spawn_agent(i as u32, base);
}

// Запускаємо координатора
manager.start_coordinator();

// Симуляція працює 30 секунд
std::thread::sleep(Duration::from_secs(30));
```

```
use std::sync::Arc;
use std::sync::atomic::{AtomicBool, Ordering};

fn main() {
    println!("==== Swarm Simulation (Ctrl+C to stop) ====");

    let world_bounds = Area::new(0.0, 0.0, 100.0, 100.0);
    let mut manager = SwarmManager::new(world_bounds);

    // Ctrl+C handler
    let shutdown = Arc::clone(&manager.shared.shutdown);
    ctrlc::set_handler(move || {
        println!("\n[Signal] Ctrl+C received, shutting down...");
        shutdown.store(true, Ordering::SeqCst);
    }).expect("Error setting Ctrl+C handler");

    // Spawn agents...
    for i in 0..5 {
        manager.spawn_agent(i, Position::new(i as f64 * 20.0, 10.0));
    }

    manager.start_coordinator();

    // Чекаємо на shutdown signal
    while !manager.shared.shutdown.load(Ordering::SeqCst) {
        std::thread::sleep(std::time::Duration::from_millis(100));
    }

    manager.shutdown();
    println!("==== Goodbye ====");
```

```
use tracing::{info, warn, error, debug, instrument};
use tracing_subscriber;

// Ініціалізація
fn init_logging() {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::DEBUG)
        .with_target(false)
        .with_thread_ids(true)
        .init();
}

// у Agent
#[instrument(skip(self))]
impl Agent {
    fn tick(&mut self) {
        debug!(agent_id = %self.id, position = %self.position, "tick");

        if self.should_return_to_base() {
            warn!(agent_id = %self.id, battery = self.battery, "low battery");
        }
    }
}

// у Coordinator
impl Coordinator {
    fn handle_message(&mut self, msg: AgentMessage) {
        match &msg {
            AgentMessage::TargetDetected { agent_id, target } => {
                info!(%agent_id, target_id = ?target.id, "target detected");
            }
        }
    }
}
```

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

#[test]
fn test_position_distance() {
    let p1 = Position::new(0.0, 0.0);
    let p2 = Position::new(3.0, 4.0);
    assert!((p1.distance_to(&p2) - 5.0).abs() < 0.001);
}

#[test]
fn test_area_contains() {
    let area = Area::new(0.0, 0.0, 10.0, 10.0);
    assert!(area.contains(&Position::new(5.0, 5.0)));
    assert!(!area.contains(&Position::new(15.0, 5.0)));
}

#[test]
fn test_agent_state_transitions() {
    let (tx, _rx) = mpsc::channel();
    let (_cmd_tx, cmd_rx) = mpsc::channel();
    let shared = SharedState::new(Area::new(0.0, 0.0, 100.0, 100.0));

    let mut agent = Agent::new(AgentId(1), Position::new(0.0, 0.0), tx, cmd_rx, shared);

    assert!(matches!(agent.state, AgentState::Idle));

    agent.handle_command(Command::Patrol { area: Area::new(0.0, 0.0, 10.0, 10.0) });
    assert!(matches!(agent.state, AgentState::Patrolling { .. }));
}
```

```
#[test]
fn test_swarm_integration() {
    let bounds = Area::new(0.0, 0.0, 50.0, 50.0);
    let mut manager = SwarmManager::new(bounds);

    // Spawn 3 agents
    manager.spawn_agent(0, Position::new(10.0, 10.0));
    manager.spawn_agent(1, Position::new(40.0, 10.0));
    manager.spawn_agent(2, Position::new(25.0, 40.0));

    manager.start_coordinator();

    // Даємо системі попрацювати
    std::thread::sleep(std::time::Duration::from_secs(2));

    // Перевіряємо статистику
    let stats = &manager.shared.stats;
    assert!(stats.total_messages.load(Ordering::Relaxed) > 0);
    assert!(stats.active_agents.load(Ordering::SeqCst) == 3);

    // Перевіряємо карту
    let map = manager.shared.world_map.read().unwrap();
    assert!(!map.explored.is_empty());

    // Shutdown
    manager.shutdown();

    assert!(stats.active_agents.load(Ordering::SeqCst) == 0);
}
```

```
swarm_simulation/
├── Cargo.toml
└── src/
    ├── main.rs          # Точка входу
    └── lib.rs           # Публічний API
    └── types/
        ├── mod.rs
        ├── position.rs   # Position, Area
        ├── agent.rs       # AgentId, AgentInfo, AgentState
        ├── target.rs      # Target, Detection
        └── messages.rs    # AgentMessage, Command
    └── agent/
        ├── mod.rs
        └── agent.rs       # Agent implementation
    └── coordinator/
        ├── mod.rs
        └── coordinator.rs
    └── world/
        ├── mod.rs
        └── map.rs          # WorldMap
    └── shared.rs         # SharedState, SwarmStats
    └── manager.rs        # SwarmManager
tests/
└── integration.rs
```

```
[package]
name = "swarm_simulation"
version = "0.1.0"
edition = "2021"

[dependencies]
# Рандом для симуляції
rand = "0.8"

# Логування
tracing = "0.1"
tracing-subscriber = "0.3"

# Ctrl+C handling
ctrlc = { version = "3.4", features = ["termination"] }

# Кращі канали (опціонально)
crossbeam-channel = "0.5"

# Сепіалізація (для збереження стану)
serde = { version = "1.0", features = ["derive"] }
serde_json = "1.0"

[dev-dependencies]
criterion = "0.5" # Benchmarks

[[bench]]
name = "swarm_bench"
harness = false
```

Можливі розширення

1. Візуалізація

- Terminal UI (ratatui)
- Web UI (WebSocket + Canvas)

2. Покращена координація

- Auction-based task allocation
- Formation control
- Collision avoidance

3. Persistence

- Збереження стану в файл
- Replay симуляції

4. Networking

- Розподілений рій (TCP/UDP)
- gRPC для комунікації

5. Machine Learning

- Reinforcement learning для поведінки
- Path planning з A*

Підсумок практикуму

Створено повноцінну мультиагентну систему:

- ✓ Agent — потік з машиною станів
- ✓ Coordinator — центральне керування
- ✓ WorldMap — спільна карта (Arc<RwLock>)
- ✓ SwarmStats — статистика (Atomic)
- ✓ Channels — комунікація Agent ↔ Coordinator
- ✓ Graceful shutdown — коректне завершення
- ✓ Testing — unit та integration тести

Використані концепції:

- Threads, Arc, Mutex, RwLock
 - mpsc channels
 - Atomic types
 - Pattern matching, enums
- [Наступна частина: Async/Await — асинхронний рій](#)
- Module system

Завдання для самостійної роботи

1. Базова реалізація:

- Запустіть код з лекції
- Додайте 10 агентів
- Спостерігайте за логами

2. Collision avoidance:

- Агенти не повинні зіштовхуватись
- Мінімальна відстань 2.0

3. Target tracking:

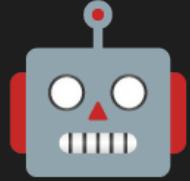
- Ціль рухається
- Агенти переслідують
- Координатор призначає найближчого

4. Formation control:

- Агенти тримають формацію (трикутник, лінія)
- Leader-follower pattern

5. Metrics та Dashboard:

- Збір метрик (prometheus format)
- Terminal dashboard з ratatui



Локальний рій готовий!

Threads • Channels • Arc • Mutex • Atomic

Практикум завершено!

Питання?