Лекція 18

Async Channels та синхронізація

Глибоке занурення в async комунікацію

Patterns • Backpressure • Fan-out • Pipeline • Actor

🤖 Архітектура комунікації для масштабованого рою

Частина 1: Патерни каналів

О. С. Бичков • 2025

План лекції (Частина 1)

1. Повторення: типи async каналів
2. Backpressure детально
3. Вибір розміру буфера
4. Channel closing семантика
5. Graceful shutdown через канали
6. select! з каналами детально
7. biased vs fair selection
8. Обробка закритих каналів
9. Fan-out pattern
10. Fan-in pattern
11. Pipeline pattern
12. Request-Response pattern
13. 🤖 Coordinator з select!
14. 🤖 Multi-channel agent
15. 🤖 Priority commands
16. Типові помилки

Частина 2: Actor model, Event Bus, складні патерни

Async канали — повторення

Канал	Send	Recv	Буфер	Clone Rx
mpsc bounded	async	async	N	❌
mpsc unbounded	sync	async	∞	❌
broadcast	sync	async	N	✓ (subscribe)
watch	sync	async	1	✓
oneshot	sync	async	1	❌

Ключові відмінності:
- mpsc — багато відправників, один отримувач
- broadcast — кожен отримує кожне повідомлення
- watch — тільки останнє значення
- oneshot — один раз, один результат

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

let (tx, mut rx) = mpsc::channel::<Data>(100);  // Buffer 100

// Producer
tokio::spawn(async move {
    loop {
        let data = generate_data();
        // send().await БЛОКУЄ якщо buffer повний!
        tx.send(data).await.unwrap();
    }
});

// Consumer (повільніший)
while let Some(data) = rx.recv().await {
    slow_processing(data).await;  // 10ms на елемент
}

// Producer автоматично сповільнюється!
```

```
// Рекомендації:

// Для команд (низький traffic, швидка обробка)
let (cmd_tx, cmd_rx) = mpsc::channel::<Command>(32);

// Для даних сенсорів (високий traffic)
let (sensor_tx, sensor_rx) = mpsc::channel::<SensorData>(1000);

// Для координатора (багато агентів)
// buffer = кількість агентів × повідомлень за tick
let num_agents = 100;
let msgs_per_tick = 2;
let (tx, rx) = mpsc::channel(num_agents * msgs_per_tick);

// Правило: buffer = N × очікувана затримка обробки
// unbounded — тільки коли впевнені що consumer швидший
let (tx, rx) = mpsc::unbounded_channel();  // ⚠️ OOM risk
```

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

let (tx, mut rx) = mpsc::channel::<i32>(10);
let tx2 = tx.clone();

// Коли закривається канал?

// 1. Всі Sender dropped → канал закритий
drop(tx);
drop(tx2);
// rx.recv().await → None

// 2. Receiver dropped → канал закритий
drop(rx);
// tx.send(42).await → Err(SendError)
// tx.is_closed() → true

// 3. Явне закриття Receiver
rx.close();
// Нові send() → Err
// Існуючі повідомлення в буфері все ще можна отримати!

// Перевірка:
if tx.is_closed() {
    println!("Receiver is gone or closed");
}

// Очікування закриття:
tx.closed().await;  // Чекає поки receiver закриється
```

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

async fn producer(tx: mpsc::Sender<Data>, mut shutdown: mpsc::Receiver<()>) {
    loop {
        tokio::select! {
            _ = shutdown.recv() => {
                println!("Producer shutting down");
                break;
            }
            _ = produce_data(&tx) => {}
        }
    }
    // tx dropped тут — сигнал consumer'y
}

async fn consumer(mut rx: mpsc::Receiver<Data>) {
    // Обробляємо ВСЕ що залишилось в буфері
    while let Some(data) = rx.recv().await {
        process(data).await;
    }
    println!("Consumer done, all data processed");
}

// Main
let (shutdown_tx, shutdown_rx) = mpsc::channel(1);
let (data_tx, data_rx) = mpsc::channel(100);

// ... spawn tasks ...

// Shutdown:
```

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

async fn multi_channel_handler(
    mut commands: mpsc::Receiver<Command>,
    mut events: mpsc::Receiver<Event>,
    mut shutdown: mpsc::Receiver<()>,
) {
    loop {
        select! {
            // Кожна гілка — окремий pattern match
            Some(cmd) = commands.recv() => {
                handle_command(cmd).await;
            }

            Some(event) = events.recv() => {
                handle_event(event).await;
            }

            _ = shutdown.recv() => {
                println!("Shutdown received");
                break;
            }

            // else — коли ВСІ канали закриті
            else => {
                println!("All channels closed");
                break;
            }
        }
```

```
use tokio::select;

// За замовчуванням — FAIR (випадковий порядок)
loop {
    select! {
        Some(a) = rx_a.recv() => { /* ... */ }
        Some(b) = rx_b.recv() => { /* ... */ }
        // Якщо обидва готові — випадковий вибір
    }
}

// biased — перевірка ПО ПОРЯДКУ
loop {
    select! {
        biased;  // ← Важливо!

        // Перевіряється ПЕРШИМ (пріоритет)
        _ = shutdown.recv() => {
            break;
        }

        // Перевіряється другим
        Some(cmd) = high_priority.recv() => {
            handle_high(cmd).await;
        }

        // Перевіряється останнім
        Some(cmd) = low_priority.recv() => {
            handle_low(cmd).await;
        }
```

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

async fn robust_handler(
    mut primary: mpsc::Receiver<Data>,
    mut backup: mpsc::Receiver<Data>,
) {
    let mut primary_closed = false;
    let mut backup_closed = false;

    loop {
        select! {
            result = primary.recv(), if !primary_closed => {
                match result {
                    Some(data) => process(data).await,
                    None => {
                        println!("Primary closed");
                        primary_closed = true;
                    }
                }
            }

            result = backup.recv(), if !backup_closed => {
                match result {
                    Some(data) => process(data).await,
                    None => {
                        println!("Backup closed");
                        backup_closed = true;
                    }
                }
            }
```

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

/// Fan-out: один producer → багато workers
async fn fan_out<T: Clone + Send + 'static>(
    mut source: mpsc::Receiver<T>,
    worker_count: usize,
) {
    // Створюємо канал для кожного worker
    let mut workers: Vec<mpsc::Sender<T>> = Vec::new();
    let mut tasks = JoinSet::new();

    for i in 0..worker_count {
        let (tx, rx) = mpsc::channel(100);
        workers.push(tx);

        tasks.spawn(async move {
            worker_loop(i, rx).await;
        });
    }

    // Round-robin розподіл
    let mut next_worker = 0;
    while let Some(item) = source.recv().await {
        workers[next_worker].send(item).await.ok();
        next_worker = (next_worker + 1) % worker_count;
    }

    // Закриваємо канали workers
    drop(workers);
```

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

/// Fan-in: багато sources → один collector
async fn fan_in<T: Send + 'static>(
    mut sources: Vec<mpsc::Receiver<T>>,
) -> mpsc::Receiver<T> {
    let (tx, rx) = mpsc::channel(sources.len() * 10);

    // Spawn task для кожного source
    for mut source in sources {
        let tx = tx.clone();
        tokio::spawn(async move {
            while let Some(item) = source.recv().await {
                if tx.send(item).await.is_err() {
                    break;  // Collector gone
                }
            }
        });
    }

    rx
}

// Використання:
let agents_rx: Vec<mpsc::Receiver<AgentReport>> = create_agents();
let mut combined = fan_in(agents_rx).await;

while let Some(report) = combined.recv().await {
    coordinator.process_report(report).await;
```

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

// Pipeline: Stage1 → Stage2 → Stage3 → Output

async fn create_pipeline() -> mpsc::Receiver<ProcessedData> {
    let (input_tx, input_rx) = mpsc::channel::<RawData>(100);
    let (stage1_tx, stage1_rx) = mpsc::channel::<Stage1Data>(50);
    let (stage2_tx, stage2_rx) = mpsc::channel::<Stage2Data>(50);
    let (output_tx, output_rx) = mpsc::channel::<ProcessedData>(50);

    // Stage 1: Валідація
    tokio::spawn(async move {
        while let Some(raw) = input_rx.recv().await {
            if let Some(valid) = validate(raw) {
                stage1_tx.send(valid).await.ok();
            }
        }
    });

    // Stage 2: Трансформація
    tokio::spawn(async move {
        while let Some(data) = stage1_rx.recv().await {
            let transformed = transform(data).await;
            stage2_tx.send(transformed).await.ok();
        }
    });

    // Stage 3: Фінальна обробка
    tokio::spawn(async move {
        while let Some(data) = stage2_rx.recv().await {
```

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

// Request містить канал для відповіді
struct Request {
    query: String,
    response_tx: oneshot::Sender<Response>,
}

struct Response {
    data: String,
}

// Service
async fn service(mut requests: mpsc::Receiver<Request>) {
    while let Some(req) = requests.recv().await {
        let result = process_query(&req.query).await;
        let _ = req.response_tx.send(Response { data: result });
    }
}

// Client
async fn client_request(service_tx: &mpsc::Sender<Request>) -> Response {
    let (response_tx, response_rx) = oneshot::channel();

    service_tx.send(Request {
        query: "SELECT * FROM agents".to_string(),
        response_tx,
    }).await.unwrap();

    // Чекаємо відповідь
```

MAS Coordinator pattern

```
struct Coordinator {
    from_agents: mpsc::Receiver<AgentMessage>,
    to_agents: HashMap<AgentId, mpsc::Sender<Command>>,
    broadcast: broadcast::Sender<BroadcastCmd>,
    state: watch::Sender<GlobalState>,
    shutdown: mpsc::Receiver<()>,
}

impl Coordinator {
    async fn run(mut self) {
        let mut tick = tokio::time::interval(Duration::from_millis(100));

        loop {
            select! {
                biased;

                _ = self.shutdown.recv() => {
                    self.shutdown_all().await;
                    break;
                }

                Some(msg) = self.from_agents.recv() => {
                    self.handle_message(msg).await;
                }

                _ = tick.tick() => {
                    self.periodic_update().await;
                }
            }
        }
```

```
struct AgentChannels {
    commands: mpsc::Receiver<Command>,         // Персональні команди
    broadcast: broadcast::Receiver<BroadcastCmd>, // Загальні команди
    state_updates: watch::Receiver<GlobalState>,  // Стан світу
    to_coordinator: mpsc::Sender<AgentMessage>,   // Звіти
}

impl Agent {
    async fn run(mut self, mut channels: AgentChannels) {
        let mut tick = tokio::time::interval(Duration::from_millis(100));

        loop {
            select! {
                biased;

                // Broadcast має пріоритет (shutdown!)
                Ok(cmd) = channels.broadcast.recv() => {
                    if matches!(cmd, BroadcastCmd::Shutdown) { break; }
                    self.handle_broadcast(cmd).await;
                }

                Some(cmd) = channels.commands.recv() => {
                    self.handle_command(cmd).await;
                }

                Ok(_) = channels.state_updates.changed() => {
                    let state = channels.state_updates.borrow().clone();
                    self.update_world_view(state);
                }
```

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

enum Priority {
    Critical,  // Негайне виконання
    High,      // Важливі команди
    Normal,    // Звичайні команди
    Low,       // Фонові задачі
}

struct PriorityChannels {
    critical: mpsc::Receiver<Command>,
    high: mpsc::Receiver<Command>,
    normal: mpsc::Receiver<Command>,
    low: mpsc::Receiver<Command>,
}

impl Agent {
    async fn process_commands(&mut self, ch: &mut PriorityChannels) {
        select! {
            biased;  // Пріоритет за порядком!

            Some(cmd) = ch.critical.recv() => {
                self.execute_immediately(cmd).await;
            }
            Some(cmd) = ch.high.recv() => {
                self.execute_soon(cmd).await;
            }
            Some(cmd) = ch.normal.recv() => {
                self.queue_command(cmd);
            }
```

MAS Message Routing Hub

```rust
use tokio::sync::mpsc;
use std::collections::HashMap;

struct MessageRouter {
    incoming: mpsc::Receiver<RoutedMessage>,
    routes: HashMap<AgentId, mpsc::Sender<Message>>,
    groups: HashMap<GroupId, Vec<AgentId>>,
}

impl MessageRouter {
    async fn run(mut self) {
        while let Some(msg) = self.incoming.recv().await {
            match msg.destination {
                Destination::Single(id) => {
                    if let Some(tx) = self.routes.get(&id) {
                        let _ = tx.send(msg.payload).await;
                    }
                }
                Destination::Group(group_id) => {
                    if let Some(members) = self.groups.get(&group_id) {
                        for id in members {
                            if let Some(tx) = self.routes.get(id) {
                                let _ = tx.send(msg.payload.clone()).await;
                            }
                        }
                    }
                }
                Destination::Broadcast => {
                    for tx in self.routes.values() {
                        let _ = tx.send(msg.payload.clone()).await;
```

Типові помилки з async каналами

```
let (tx, rx) = mpsc::unbounded();
// Producer швидший → OOM
```

✓ Bounded з backpressure

```
let (tx, rx) = mpsc::channel(100);
// Backpressure!
```

Ще типові помилки

```
select! {
    x = rx.recv() => { } // None?
}
```

✓ 3 else для завершення

```
select! {
    Some(x) = rx.recv() => { }
    else => break
}
```

Best Practices: Channels

✓ Bounded channels для production (backpressure)
✓ biased select! для пріоритетів (shutdown першим!)
✓ Handle всі варіанти закриття каналів
✓ Graceful shutdown: signal → drain buffer → exit
✓ Розмір буфера = traffic × latency

❌ Не використовуйте unbounded без вагомої причини
❌ Не ігноруйте Lagged у broadcast
❌ Не забувайте drop sender для закриття
❌ Не блокуйте в обробниках (spawn для важкої роботи)

🤖 Для MAC:
• mpsc для Agent → Coordinator
• broadcast для Coordinator → All Agents
• watch для shared state
• biased select! для graceful shutdown

Підсумок: Частина 1

Backpressure:
- Bounded channels автоматично балансують
- Вибір буфера: traffic × expected latency

Channel closing:
- Drop всіх Sender → Receiver gets None
- rx.close() — м'яке закриття

select!:
- biased для пріоритетів
- else для обробки закритих каналів
- if guard для умовного вмикання

Patterns:
- Fan-out/Fan-in для паралельної обробки
- Pipeline для послідовної обробки
- Request-Response з oneshot

→ Частина 2: Actor model, Event Bus, складні патерни

Лекція 18 (продовження)

Actor Model та Event Bus

Архітектурні патерни для async систем

Actor • Mailbox • Event Bus • Supervisor • Registry

🤖 Production-ready архітектура для MAC

Частина 2: Складні патерни

О. С. Бичков • 2025

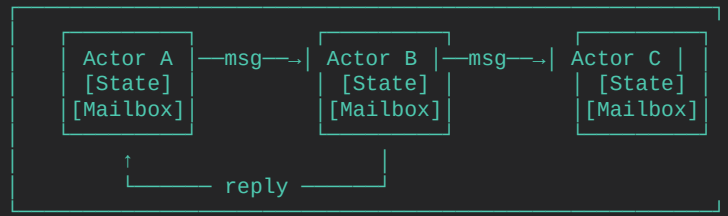
План лекції (Частина 2)

1. Actor Model — концепція
2. Actor на Tokio
3. Actor Handle pattern
4. Actor Mailbox
5. Typed messages
6. Request-Reply в Actor
7. Actor Supervisor
8. Actor Registry
9. Event Bus pattern
10. Pub/Sub система
11. 🤖 Agent як Actor
12. 🤖 Coordinator Actor
13. 🤖 Swarm Event Bus
14. 🤖 Full Async Swarm
15. Performance tips
16. Підсумок

Actor Model — концепція

Actor Model — парадигма конкурентного програмування:

- Кожен Actor — ізольована одиниця з власним станом
- Actors спілкуються ТІЛЬКИ через повідомлення
- Кожен Actor обробляє повідомлення послідовно
- Actors можуть створювати інших Actors
- Ніякого спільного стану — немає data races!



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

// Actor = Task + Channel (mailbox)
struct MyActor {
    receiver: mpsc::Receiver<ActorMessage>,
    state: i32,
}

enum ActorMessage {
    Increment,
    GetValue { respond_to: oneshot::Sender<i32> },
}

impl MyActor {
    fn new(receiver: mpsc::Receiver<ActorMessage>) -> Self {
        MyActor { receiver, state: 0 }
    }

    async fn run(mut self) {
        while let Some(msg) = self.receiver.recv().await {
            self.handle_message(msg);
        }
    }

    fn handle_message(&mut self, msg: ActorMessage) {
        match msg {
            ActorMessage::Increment => self.state += 1,
            ActorMessage::GetValue { respond_to } => {
                let _ = respond_to.send(self.state);
            }
```

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

// Handle — публічний інтерфейс до Actor
#[derive(Clone)]
pub struct MyActorHandle {
    sender: mpsc::Sender<ActorMessage>,
}

impl MyActorHandle {
    pub fn new() -> Self {
        let (sender, receiver) = mpsc::channel(100);
        let actor = MyActor::new(receiver);
        tokio::spawn(actor.run());
        MyActorHandle { sender }
    }

    pub async fn increment(&self) {
        let _ = self.sender.send(ActorMessage::Increment).await;
    }

    pub async fn get_value(&self) -> i32 {
        let (send, recv) = oneshot::channel();
        let _ = self.sender.send(ActorMessage::GetValue { respond_to: send }).await;
        recv.await.expect("Actor dropped")
    }
}

// Використання:
let actor = MyActorHandle::new();
actor.increment().await;
```

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

// Кожна команда — окремий тип з response type
pub enum AgentCommand {
    MoveTo {
        position: Position,
        respond_to: oneshot::Sender<MoveResult>,
    },
    Scan {
        radius: f64,
        respond_to: oneshot::Sender<Vec<Target>>,
    },
    GetStatus {
        respond_to: oneshot::Sender<AgentStatus>,
    },
    Shutdown,
}

// Actor Handle з типізованими методами
impl AgentHandle {
    pub async fn move_to(&self, pos: Position) -> Result<MoveResult, Error> {
        let (tx, rx) = oneshot::channel();
        self.sender.send(AgentCommand::MoveTo {
            position: pos,
            respond_to: tx,
        }).await?;
        Ok(rx.await?)
    }

    pub async fn scan(&self, radius: f64) -> Result<Vec<Target>, Error> {
```

```rust
// Bounded mailbox — з backpressure
pub struct BoundedActor {
    mailbox: mpsc::Receiver<Message>,  // bounded
}

impl BoundedActorHandle {
    pub fn new(capacity: usize) -> Self {
        let (tx, rx) = mpsc::channel(capacity);  // Backpressure!
        // ...
    }

    // send().await може чекати якщо mailbox повний
    pub async fn send(&self, msg: Message) -> Result<(), Error> {
        self.sender.send(msg).await.map_err(|_| Error::ActorDead)
    }

    // try_send — non-blocking, може відхилити
    pub fn try_send(&self, msg: Message) -> Result<(), TrySendError> {
        self.sender.try_send(msg)
    }
}

// Unbounded — для критичних повідомлень
pub struct UnboundedActor {
    mailbox: mpsc::UnboundedReceiver<CriticalMessage>,
}

// ⚠️ Unbounded ризик OOM якщо producer швидший!
```

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

impl AgentHandle {
    /// Запит зі стандартним timeout
    pub async fn request<R>(
        &self,
        make_msg: impl FnOnce(oneshot::Sender<R>) -> AgentCommand,
    ) -> Result<R, AgentError> {
        let (tx, rx) = oneshot::channel();

        // Надсилаємо запит
        self.sender.send(make_msg(tx)).await
            .map_err(|_| AgentError::ActorDead)?;

        // Чекаємо відповідь з timeout
        timeout(Duration::from_secs(5), rx).await
            .map_err(|_| AgentError::Timeout)?
            .map_err(|_| AgentError::ActorDead)
    }

    pub async fn get_status(&self) -> Result<AgentStatus, AgentError> {
        self.request(|respond_to| AgentCommand::GetStatus { respond_to }).await
    }

    pub async fn move_to(&self, pos: Position) -> Result<MoveResult, AgentError> {
        self.request(|respond_to| AgentCommand::MoveTo {
            position: pos,
            respond_to,
        }).await
    }
```

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

struct Supervisor {
    children: JoinSet<ActorResult>,
    actor_configs: HashMap<ActorId, ActorConfig>,
}

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(ActorResult::Completed(id)) => {
                        println!("Actor {} completed normally", id);
                    }
                    Ok(ActorResult::Failed(id, error)) => {
                        println!("Actor {} failed: {:?}, restarting...", id, error);
                        self.restart_actor(id).await;
                    }
                    Err(join_error) => {
                        // Actor panicked
                        println!("Actor panicked: {:?}", join_error);
                    }
                }
            } else {
                break;  // Всі actors завершились
            }
        }
    }
```

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

/// Глобальний реєстр акторів
pub struct ActorRegistry {
    agents: RwLock<HashMap<AgentId, AgentHandle>>,
    groups: RwLock<HashMap<GroupId, Vec<AgentId>>>,
}

impl ActorRegistry {
    pub fn new() -> Arc<Self> {
        Arc::new(ActorRegistry {
            agents: RwLock::new(HashMap::new()),
            groups: RwLock::new(HashMap::new()),
        })
    }

    pub async fn register(&self, id: AgentId, handle: AgentHandle) {
        self.agents.write().await.insert(id, handle);
    }

    pub async fn unregister(&self, id: AgentId) {
        self.agents.write().await.remove(&id);
    }

    pub async fn get(&self, id: AgentId) -> Option<AgentHandle> {
        self.agents.read().await.get(&id).cloned()
    }
```

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

#[derive(Clone, Debug)]
pub enum SwarmEvent {
    AgentSpawned(AgentId),
    AgentStopped(AgentId),
    TargetDetected { by: AgentId, target: Target },
    MissionStarted(MissionId),
    MissionCompleted(MissionId),
    Alert(AlertLevel, String),
}

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

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

    pub fn publish(&self, event: SwarmEvent) {
        let _ = self.sender.send(event);  // Ignore if no receivers
    }

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

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

impl EventBus {
    /// Підписка з фільтром
    pub fn subscribe_filtered<F>(
        &self,
        mut filter: F,
    ) -> impl futures::Stream<Item = SwarmEvent>
    where
        F: FnMut(&SwarmEvent) -> bool + Send + 'static,
    {
        let mut rx = self.sender.subscribe();

        async_stream::stream! {
            loop {
                match rx.recv().await {
                    Ok(event) if filter(&event) => yield event,
                    Ok(_) => continue,  // Filtered out
                    Err(broadcast::error::RecvError::Lagged(n)) => {
                        log::warn!("Missed {} events", n);
                    }
                    Err(broadcast::error::RecvError::Closed) => break,
                }
            }
        }
    }
}

// Використання:
let target_events = event_bus.subscribe_filtered(|e| {
```

```rust
use std::collections::HashMap;
use tokio::sync::{broadcast, RwLock};

pub struct TopicBus {
    topics: RwLock<HashMap<String, broadcast::Sender<Message>>>,
}

impl TopicBus {
    pub async fn publish(&self, topic: &str, msg: Message) {
        let topics = self.topics.read().await;
        if let Some(sender) = topics.get(topic) {
            let _ = sender.send(msg);
        }
    }

    pub async fn subscribe(&self, topic: &str) -> broadcast::Receiver<Message> {
        let mut topics = self.topics.write().await;
        topics.entry(topic.to_string())
            .or_insert_with(|| broadcast::channel(100).0)
            .subscribe()
    }
}

// Використання:
let mut alerts = bus.subscribe("alerts").await;
let mut targets = bus.subscribe("targets").await;

// Публікація
bus.publish("alerts", Message::Alert("Warning!".into())).await;
bus.publish("targets", Message::Target(target)).await;
```

MAC Agent Architecture

```rust
pub struct AgentActor {
    id: AgentId,
    mailbox: mpsc::Receiver<AgentCommand>,
    event_bus: Arc<EventBus>,
    state: AgentState,
}

impl AgentActor {
    async fn run(mut self) {
        self.event_bus.publish(SwarmEvent::AgentSpawned(self.id));

        let mut tick = tokio::time::interval(Duration::from_millis(100));
        let mut events = self.event_bus.subscribe();

        loop {
            select! {
                biased;

                Some(cmd) = self.mailbox.recv() => {
                    if self.handle_command(cmd).await.is_break() {
                        break;
                    }
                }

                Ok(event) = events.recv() => {
                    self.handle_event(event).await;
                }

                _ = tick.tick() => {
                    self.tick().await;
```

⚓ MAC Coordinator Actor

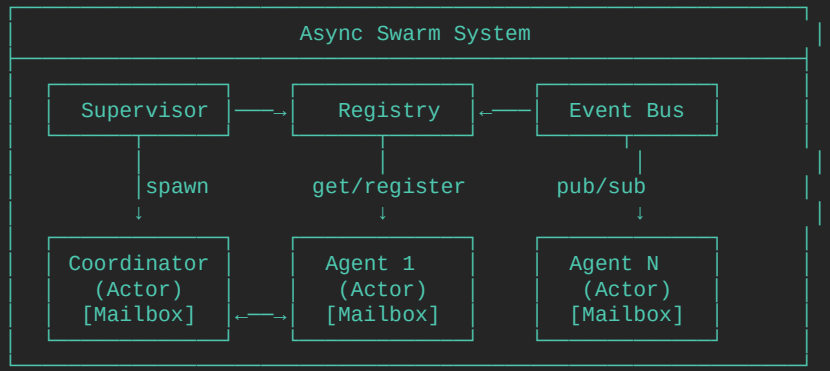
```rust
pub struct CoordinatorActor {
    mailbox: mpsc::Receiver<CoordCommand>,
    registry: Arc<ActorRegistry>,
    event_bus: Arc<EventBus>,
    state: CoordinatorState,
}

impl CoordinatorActor {
    async fn run(mut self) {
        let mut events = self.event_bus.subscribe();

        loop {
            select! {
                Some(cmd) = self.mailbox.recv() => {
                    self.handle_command(cmd).await;
                }

                Ok(event) = events.recv() => {
                    match event {
                        SwarmEvent::TargetDetected { by, target } => {
                            self.assign_target(target).await;
                        }
                        SwarmEvent::AgentStopped(id) => {
                            self.handle_agent_lost(id).await;
                        }
                        _ => {}
                    }
                }
            }
        }
```

MAC: Повна архітектура Async Swarm



```rust
#[tokio::main]
async fn main() {
    // Ініціалізація інфраструктури
    let event_bus = EventBus::new(1000);
    let registry = ActorRegistry::new();

    // Запуск Coordinator
    let coordinator = CoordinatorActor::spawn(
        Arc::clone(&registry),
        Arc::clone(&event_bus),
    );

    // Supervisor для агентів
    let supervisor = Supervisor::new(
        Arc::clone(&registry),
        Arc::clone(&event_bus),
    );

    // Spawn агентів
    for i in 0..100 {
        supervisor.spawn_agent(AgentId(i), AgentConfig::default()).await;
    }

    // Запуск supervisor
    let supervisor_handle = tokio::spawn(supervisor.run());

    // Graceful shutdown на Ctrl+C
    tokio::signal::ctrl_c().await.unwrap();
    event_bus.publish(SwarmEvent::Alert(AlertLevel::Critical, "Shutdown".into()));
```

🤖 MAC: Потік повідомлень у системі

1. Agent виявляє ціль:
   → Agent публікує TargetDetected в EventBus

2. Coordinator отримує подію:
   → Coordinator підписаний на EventBus
   → Обробляє TargetDetected

3. Coordinator призначає агента:
   → Lookup в Registry: find nearest agent
   → Send command через AgentHandle

4. Agent отримує команду:
   → Mailbox: PursueTarget command
   → Змінює стан, починає переслідування

Loose coupling через EventBus + Direct commands через Mailbox

5. Agent повідомляє про результат:
   → Публікує MissionCompleted в EventBus
   → Coordinator та інші отримують подію

```
// Batch processing
while let Ok(msg) = rx.try_recv() {
    batch.push(msg);
    if batch.len() >= 100 { break; }
}
process_batch(batch).await;
```

- Context switch: ~200-500ns

```rust
// 1. Tracing для async
use tracing::{info, instrument};

#[instrument(skip(self))]
async fn handle_message(&mut self, msg: Message) {
    info!(agent_id = %self.id, "Processing message");
    // ...
}

// 2. tokio-console для runtime inspection
// Cargo.toml: tokio = { features = ["tracing"] }
// console_subscriber::init();

// 3. Channel metrics
struct MeteredSender<T> {
    inner: mpsc::Sender<T>,
    sent_count: Arc<AtomicU64>,
}

impl<T> MeteredSender<T> {
    async fn send(&self, msg: T) -> Result<(), SendError<T>> {
        self.sent_count.fetch_add(1, Ordering::Relaxed);
        self.inner.send(msg).await
    }
}

// 4. Deadlock detection
// Якщо select! зависає — один з каналів не прогресує
// Додайте timeout branch для діагностики
```

Порівняння підходів

Підхід	Pros	Cons	Use Case
Direct channels	Simple, fast	Tight coupling	1:1 communication
Actor model	Encapsulation	Boilerplate	Complex state
Event Bus	Loose coupling	No guarantees	Notifications
Pub/Sub topics	Flexible routing	Complexity	Large systems

🤖 Рекомендація для MAC:

- Actor model для агентів (state encapsulation)
- Event Bus для системних подій
- Direct channels для Coordinator → Agent commands
- Registry для discovery

Підсумок лекції

Actor Model:
- Actor = Task + Mailbox (channel)
- Handle pattern для публічного API
- Supervisor для restart failed actors
- Registry для actor discovery

Event Bus:
- broadcast для pub/sub
- Loose coupling між компонентами
- Фільтрація подій

🤖 MAC Architecture:
- Agents як Actors з mailboxes
- Coordinator як центральний Actor
- EventBus для системних подій
- Registry для lookup агентів

→ Наступна лекція: Streams — async ітератори

Завдання для самостійної роботи

1. Simple Actor:
   - Counter actor з Increment/GetValue
   - Handle з typed methods
   - Timeout для requests

2. Event Bus:
   - Broadcast channel
   - 5 subscribers
   - Filtered subscriptions

3. Actor Registry:
   - Register/Unregister actors
   - Lookup by ID
   - Send to group

4. Agent Actor:
   - State machine в actor
   - Event publishing
   - Command handling

5. Full Swarm:
   - Coordinator + 10 Agent actors
   - Event Bus
   - Registry
   - Supervisor з restart

Async Channels опановано!

Actor • Event Bus • Supervisor • Registry

Питання?