Лекція 14

Channels: Message Passing

Комунікація через передачу повідомлень

mpsc • Sender • Receiver • sync_channel • crossbeam

🤖 Приклади: комунікація агентів, координатор рою

Частина 1: std::sync::mpsc

О. С. Бичков • 2025

План лекції (Частина 1)

1. Message Passing — філософія
2. Shared State vs Message Passing
3. mpsc — Multi-Producer Single-Consumer
4. channel() — створення каналу
5. Sender<T> — відправник
6. Receiver<T> — отримувач
7. Ownership через канали
8. send() та SendError
9. recv() та RecvError
10. try_recv() — неблокуючий
11. recv_timeout()
12. Ітерація по каналу
13. Multiple producers
14. 🤖 Агент → Координатор
15. 🤖 Broadcast повідомлень
16. Типові помилки

Частина 2: sync_channel, crossbeam, патерни

Message Passing — філософія

"Do not communicate by sharing memory;
instead, share memory by communicating."
— Go Proverb (актуально і для Rust!)

Ідея: замість спільного стану з locks,
потоки обмінюються повідомленнями через канали.

Shared State

```
Потік A ──┐
          ├─→ [Data + Lock]
Потік B ──┘
```

Обидва мають доступ до даних
Потрібна синхронізація

Message Passing

```
Потік A ─[msg]─→ Channel ─→ Потік B
```

Дані передаються через канал
Ownership переходить

Shared State vs Message Passing

Аспект	Shared State	Message Passing
Модель	Спільна пам'ять + locks	Канали + ownership
Data races	Можливі (runtime)	Неможливі (compile)
Deadlocks	Можливі	Можливі*
Складність	Locks, guards	Протоколи, буфери
Performance	Низький overhead	Копіювання даних
Масштабування	Складне	Добре
Тестування	Складне	Простіше

* Deadlock можливий: A чекає msg від B, B чекає msg від A

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

//              Sender<T>    Receiver<T>
let (tx, rx) = mpsc::channel::<String>();

// tx — можна клонувати (multi-producer)
let tx2 = tx.clone();

// rx — не можна клонувати (single-consumer)
// let rx2 = rx.clone();  // ❌ Error: Receiver doesn't impl Clone

// Типова схема:
// Producer 1 ──tx.clone()──┐
//                          ├──→ [Channel] ──→ rx ──→ Consumer
// Producer 2 ──tx.clone()──┘
```

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

fn main() {
    // channel() створює unbounded асинхронний канал
    // Тип виводиться з використання, або вказується явно
    let (tx, rx) = mpsc::channel::<i32>();

    // Відправник у новому потоці
    thread::spawn(move || {
        tx.send(42).unwrap();
        println!("Sent!");
    });

    // Отримувач у головному потоці
    let value = rx.recv().unwrap();
    println!("Received: {}", value);  // 42
}

// Unbounded = необмежений буфер
// send() ніколи не блокує (поки є пам'ять)
// recv() блокує поки є повідомлення або канал закритий
```

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

// Sender<T> — половина каналу для відправки
// impl Clone — можна мати багато відправників
// impl Send — можна передати в інший потік

let (tx, rx) = mpsc::channel::<String>();

// send() — відправити повідомлення
// Ownership переходить у канал!
let message = String::from("hello");
tx.send(message).unwrap();
// println!("{}", message);  // ❌ Error: value moved

// send() повертає Result<(), SendError<T>>
// Err якщо Receiver dropped (канал закритий)
match tx.send(String::from("world")) {
    Ok(()) => println!("Sent successfully"),
    Err(e) => println!("Failed: receiver dropped. Message: {}", e.0),
}

// Clone для multiple producers
let tx2 = tx.clone();
let tx3 = Sender::clone(&tx);  // Explicit style
```

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

// Receiver<T> — половина каналу для отримання
// НЕ Clone — тільки один отримувач
// impl Send — можна передати в інший потік

let (tx, rx) = mpsc::channel::<i32>();

// Методи Receiver:

// recv() — блокує до отримання або закриття
let value = rx.recv().unwrap();

// try_recv() — не блокує, повертає одразу
match rx.try_recv() {
    Ok(v) => println!("Got: {}", v),
    Err(mpsc::TryRecvError::Empty) => println!("No message"),
    Err(mpsc::TryRecvError::Disconnected) => println!("Closed"),
}

// recv_timeout() — блокує з timeout
use std::time::Duration;
match rx.recv_timeout(Duration::from_secs(1)) {
    Ok(v) => println!("Got: {}", v),
    Err(mpsc::RecvTimeoutError::Timeout) => println!("Timeout!"),
    Err(mpsc::RecvTimeoutError::Disconnected) => println!("Closed"),
}
```

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

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let data = vec![1, 2, 3, 4, 5];

        // Ownership переходить у канал
        tx.send(data).unwrap();

        // data більше не доступна тут!
        // println!("{:?}", data);  // ❌ Error: borrow of moved value
    });

    // Ownership переходить до отримувача
    let received = rx.recv().unwrap();
    println!("Received: {:?}", received);  // [1, 2, 3, 4, 5]

    // received тепер належить головному потоку
}

// Це гарантує відсутність data races!
// Після send() тільки receiver має доступ до даних
```

```rust
use std::sync::mpsc::{self, SendError};

let (tx, rx) = mpsc::channel::<String>();

// Drop receiver — закриваємо канал
drop(rx);

// Тепер send() поверне помилку
let result = tx.send(String::from("hello"));

match result {
    Ok(()) => println!("Sent!"),
    Err(SendError(msg)) => {
        // SendError містить повідомлення, що не вдалось надіслати
        // Можемо його відновити!
        println!("Failed to send: {}", msg);
    }
}

// SendError<T> — обгортка над T
// impl Debug, Display, Error
// Метод into_inner() — отримати T назад

if let Err(e) = tx.send(String::from("hello")) {
    let recovered: String = e.into_inner();
    println!("Recovered message: {}", recovered);
}
```

```rust
use std::sync::mpsc::{self, RecvError};

let (tx, rx) = mpsc::channel::<i32>();

// Надсилаємо одне повідомлення
tx.send(42).unwrap();

// Drop sender — закриваємо канал з боку відправника
drop(tx);

// Перший recv() успішний
println!("{}", rx.recv().unwrap());  // 42

// Другий recv() — помилка, канал порожній і закритий
match rx.recv() {
    Ok(v) => println!("Got: {}", v),
    Err(RecvError) => println!("Channel closed, no more messages"),
}

// RecvError — unit struct, не містить даних
// Означає: канал закритий і порожній

// Важливо розрізняти:
// - Канал порожній, але sender живий → recv() блокує
// - Канал закритий (sender dropped) → recv() = Err
```

```rust
use std::sync::mpsc::{self, TryRecvError};
use std::thread;
use std::time::Duration;

let (tx, rx) = mpsc::channel();

thread::spawn(move || {
    thread::sleep(Duration::from_millis(500));
    tx.send(42).unwrap();
});

// Polling loop
loop {
    match rx.try_recv() {
        Ok(value) => {
            println!("Received: {}", value);
            break;
        }
        Err(TryRecvError::Empty) => {
            // Канал порожній, але відкритий
            println!("No message yet, doing other work...");
            thread::sleep(Duration::from_millis(100));
        }
        Err(TryRecvError::Disconnected) => {
            // Канал закритий
            println!("Channel closed");
            break;
        }
    }
}
```

```rust
use std::sync::mpsc::{self, RecvTimeoutError};
use std::time::Duration;

let (tx, rx) = mpsc::channel::<String>();

// recv_timeout — чекаємо повідомлення з обмеженням часу
let timeout = Duration::from_secs(2);

match rx.recv_timeout(timeout) {
    Ok(msg) => println!("Received: {}", msg),
    Err(RecvTimeoutError::Timeout) => {
        println!("No message received within {} seconds", timeout.as_secs());
    }
    Err(RecvTimeoutError::Disconnected) => {
        println!("Channel closed before receiving message");
    }
}

// recv_deadline — абсолютний час
use std::time::Instant;
let deadline = Instant::now() + Duration::from_secs(5);
match rx.recv_deadline(deadline) {
    Ok(msg) => println!("Got: {}", msg),
    Err(e) => println!("Error: {:?}", e),
}
```

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

let (tx, rx) = mpsc::channel();

thread::spawn(move || {
    for i in 1..=5 {
        tx.send(i).unwrap();
        thread::sleep(std::time::Duration::from_millis(100));
    }
    // tx dropped тут — канал закривається
});

// Receiver реалізує IntoIterator!
// Ітерація продовжується поки канал відкритий
for value in rx {
    println!("Received: {}", value);
}
println!("Channel closed, iteration complete");

// Еквівалент:
// while let Ok(value) = rx.recv() {
//     println!("Received: {}", value);
// }

// Або explicit iterator:
// for value in rx.iter() { ... }
// rx.try_iter() — non-blocking iterator
```

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

let (tx, rx) = mpsc::channel();

// Надсилаємо кілька повідомлень
tx.send(1).unwrap();
tx.send(2).unwrap();
tx.send(3).unwrap();

// try_iter() — повертає всі доступні зараз повідомлення
// Не блокує, завершується коли буфер порожній
let messages: Vec<i32> = rx.try_iter().collect();
println!("{:?}", messages);  // [1, 2, 3]

// Корисно для batch processing
loop {
    // Обробляємо всі накопичені повідомлення
    for msg in rx.try_iter() {
        process(msg);
    }

    // Робимо іншу роботу
    do_other_work();

    thread::sleep(Duration::from_millis(100));
}
```

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

let (tx, rx) = mpsc::channel();

// Створюємо кілька producers
let mut handles = vec![];

for id in 0..3 {
    let tx_clone = tx.clone();  // Clone для кожного producer

    handles.push(thread::spawn(move || {
        for i in 0..3 {
            let msg = format!("Producer {} message {}", id, i);
            tx_clone.send(msg).unwrap();
        }
        // tx_clone dropped
    }));
}

// ВАЖЛИВО: Drop оригінальний tx!
drop(tx);  // Інакше канал ніколи не закриється

// Consumer отримує всі повідомлення
for msg in rx {
    println!("{}", msg);
}

for h in handles { h.join().unwrap(); }
```

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

let (tx, rx) = mpsc::channel();

let tx2 = tx.clone();
thread::spawn(move || {
    tx2.send("from thread").unwrap();
});

// ❌ ПОМИЛКА: Забули drop(tx)
// Оригінальний tx все ще живий!

for msg in rx {
    println!("{}", msg);
    // Виведе "from thread"
    // Потім ЗАВИСНЕ! rx.recv() чекає вічно
    // бо tx все ще існує
}

// ✓ ПРАВИЛЬНО:
drop(tx);  // Закриваємо оригінальний sender

for msg in rx {
    println!("{}", msg);
}
// Тепер ітерація завершиться коли всі tx dropped
```

Правило: drop() всі Sender коли завершили надсилання!

```rust
use std::sync::mpsc::{self, Sender};
use std::thread;

#[derive(Debug)]
enum AgentMessage {
    Position { id: u32, x: f64, y: f64 },
    TargetFound { id: u32, target: TargetInfo },
    LowBattery { id: u32, level: u8 },
    RequestHelp { id: u32, reason: String },
}

struct Agent {
    id: u32,
    to_coordinator: Sender<AgentMessage>,
}

impl Agent {
    fn report_position(&self, x: f64, y: f64) {
        self.to_coordinator.send(AgentMessage::Position {
            id: self.id, x, y
        }).unwrap();
    }

    fn report_target(&self, target: TargetInfo) {
        self.to_coordinator.send(AgentMessage::TargetFound {
            id: self.id, target
        }).unwrap();
    }
}
```

MAC Broadcaster (каскад)

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

/// Менеджер broadcast — тримає список отримувачів
struct Broadcaster<T: Clone> {
    subscribers: Vec<Sender<T>>,
}

impl<T: Clone> Broadcaster<T> {
    fn new() -> Self {
        Broadcaster { subscribers: Vec::new() }
    }

    /// Додати підписника, повертає Receiver
    fn subscribe(&mut self) -> Receiver<T> {
        let (tx, rx) = mpsc::channel();
        self.subscribers.push(tx);
        rx
    }

    /// Надіслати всім підписникам
    fn broadcast(&mut self, message: T) {
        // Видаляємо disconnected subscribers
        self.subscribers.retain(|tx| {
            tx.send(message.clone()).is_ok()
        });
    }
}

// Використання
let mut broadcaster = Broadcaster::new();
```

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

/// Канали для двонаправленої комунікації
struct AgentChannels {
    to_coordinator: Sender<AgentMessage>,
    from_coordinator: Receiver<Command>,
}

struct CoordinatorChannels {
    from_agents: Receiver<AgentMessage>,
    to_agents: Vec<Sender<Command>>,
}

fn create_agent_connection() -> (AgentChannels, Sender<Command>, Receiver<AgentMessage>) {
    let (agent_tx, coord_rx) = mpsc::channel();  // Agent → Coordinator
    let (coord_tx, agent_rx) = mpsc::channel();  // Coordinator → Agent

    let agent_channels = AgentChannels {
        to_coordinator: agent_tx,
        from_coordinator: agent_rx,
    };

    (agent_channels, coord_tx, coord_rx)
}

// Агент
impl Agent {
    fn run(&self) {
        loop {
            // Перевіряємо команди від координатора
```

Типові помилки з channels

```
drop(rx);
tx.send(42).unwrap(); // Panic!
```

✓ Перевіряйте Result

```
if tx.send(42).is_err() {
    println!("Receiver gone");
}
```

Ще типові помилки

```
// Producer швидший за consumer
// Буфер росте необмежено → OOM
```

✓ sync_channel з буфером

```
let (tx, rx) = sync_channel(100);
// Backpressure!
```

Підсумок: Частина 1

Message Passing — альтернатива shared state:
• Ownership передається через канали
• Компілятор гарантує відсутність data races

std::sync::mpsc:
• channel() — unbounded async канал
• Sender<T> — Clone, можна багато
• Receiver<T> — не Clone, тільки один

Методи:
• send() / recv() — блокуючі
• try_recv() — non-blocking
• recv_timeout() — з timeout
• for msg in rx — ітерація

→ Частина 2: sync_channel, crossbeam, патерни

⚠️ Не забувайте drop(tx) для закриття каналу!

Лекція 14 (продовження)

sync_channel та crossbeam

Bounded channels та розширені можливості

sync_channel • crossbeam • select • oneshot

🤖 Приклади: worker pool, request-response, event bus

Частина 2: Розширені канали

О. С. Бичков • 2025

План лекції (Частина 2)

1. sync_channel — bounded channel
2. Backpressure
3. sync_channel(0) — rendezvous
4. crossbeam-channel crate
5. crossbeam::select!
6. bounded vs unbounded
7. oneshot channel
8. Broadcast channel
9. 🤖 Worker Pool
10. 🤖 Request-Response
11. 🤖 Event Bus
12. 🤖 Pipeline processing
13. 🤖 Fan-out / Fan-in
14. Channel patterns
15. Performance tips
16. Коли channels, коли mutex

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

// sync_channel(n) — канал з обмеженим буфером
// send() БЛОКУЄ якщо буфер повний!

let (tx, rx) = mpsc::sync_channel::<i32>(2);  // Буфер на 2 елементи

tx.send(1).unwrap();  // ✓ Буфер: [1]
tx.send(2).unwrap();  // ✓ Буфер: [1, 2] — повний!

// Наступний send() заблокує поки rx не забере!
// tx.send(3).unwrap();  // ⏳ БЛОКУЄ!

// В іншому потоці:
let val = rx.recv().unwrap();  // Забираємо 1
// Тепер буфер: [2] — є місце
// send(3) розблокується

// Різниця:
// channel()      — unbounded, send() ніколи не блокує
// sync_channel() — bounded, send() блокує коли повний
```

sync_channel забезпечує backpressure — producer не може переповнити буфер

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

// Producer генерує 1000 msg/sec
// Consumer обробляє 100 msg/sec

let (tx, rx) = sync_channel(10);  // Буфер 10

thread::spawn(move || {
    for i in 0.. {
        tx.send(i).unwrap();  // Блокує коли буфер повний!
        // Producer автоматично сповільнюється
    }
});

for msg in rx {
    slow_processing(msg);  // 10ms на повідомлення
}

// Буфер ніколи не перевищить 10 елементів!
```

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

// sync_channel(0) — буфер розміром 0!
// send() блокує поки receive() не готовий
// Синхронна передача

let (tx, rx) = sync_channel::<String>(0);

let sender = thread::spawn(move || {
    println!("Sending...");
    tx.send("Hello".to_string()).unwrap();  // БЛОКУЄ!
    println!("Sent!");  // Тільки після recv()
});

thread::sleep(std::time::Duration::from_secs(1));
println!("About to receive...");
let msg = rx.recv().unwrap();  // Розблоковує send()
println!("Received: {}", msg);

sender.join().unwrap();

// Вивід:
// Sending...
// (пауза 1 сек)
// About to receive...
// Sent!
// Received: Hello
```

```rust
use std::sync::mpsc::{sync_channel, TrySendError};

let (tx, rx) = sync_channel::<i32>(2);

tx.send(1).unwrap();
tx.send(2).unwrap();  // Буфер повний

// try_send() — не блокує, повертає помилку
match tx.try_send(3) {
    Ok(()) => println!("Sent!"),
    Err(TrySendError::Full(msg)) => {
        // Буфер повний, повідомлення повернуто
        println!("Buffer full, message {} not sent", msg);
    }
    Err(TrySendError::Disconnected(msg)) => {
        // Receiver dropped
        println!("Receiver gone, message {} lost", msg);
    }
}

// Корисно для non-blocking producers
if tx.try_send(value).is_err() {
    // Буфер повний — можемо:
    // - Відкинути повідомлення
    // - Зберегти для пізніше
    // - Логувати warning
}
```

```rust
// crossbeam-channel — популярна альтернатива std::sync::mpsc
// Cargo.toml: crossbeam-channel = "0.5"

use crossbeam_channel::{unbounded, bounded, select};

// Переваги над std::sync::mpsc:
// • MPMC — multi-producer multi-consumer!
// • select! макрос для очікування на кількох каналах
// • Швидший
// • Більше типів каналів

// Unbounded
let (tx, rx) = unbounded::<i32>();

// Bounded
let (tx, rx) = bounded::<i32>(10);

// Receiver можна клонувати!
let rx2 = rx.clone();  // ✓ MPMC!

// Два consumers
std::thread::spawn(move || {
    for msg in rx { println!("Consumer 1: {}", msg); }
});
std::thread::spawn(move || {
    for msg in rx2 { println!("Consumer 2: {}", msg); }
});
```

```rust
use crossbeam_channel::{unbounded, select, Receiver};
use std::time::Duration;

let (tx1, rx1) = unbounded::<String>();
let (tx2, rx2) = unbounded::<i32>();

// select! — чекає на БУДЬ-ЯКИЙ з каналів
loop {
    select! {
        recv(rx1) -> msg => {
            match msg {
                Ok(s) => println!("String: {}", s),
                Err(_) => println!("rx1 closed"),
            }
        }
        recv(rx2) -> msg => {
            match msg {
                Ok(n) => println!("Number: {}", n),
                Err(_) => println!("rx2 closed"),
            }
        }
        // Timeout — якщо ніхто не готовий
        default(Duration::from_secs(1)) => {
            println!("No message in 1 second");
        }
    }
}
```

```
use crossbeam_channel::{bounded, select};

let (tx1, rx1) = bounded::<i32>(1);
let (tx2, rx2) = bounded::<i32>(1);

// select! може чекати і на send, і на recv
let value = 42;

select! {
    // Надіслати в перший вільний канал
    send(tx1, value) -> res => {
        match res {
            Ok(()) => println!("Sent to channel 1"),
            Err(_) => println!("Channel 1 closed"),
        }
    }
    send(tx2, value) -> res => {
        match res {
            Ok(()) => println!("Sent to channel 2"),
            Err(_) => println!("Channel 2 closed"),
        }
    }
}

// Корисно для load balancing
// Надсилаємо в канал, який готовий прийняти
```

Порівняння типів каналів

Канал	Producers	Consumers	Buffer	select!
mpsc::channel	Multi	Single	Unbounded	❌
mpsc::sync_channel	Multi	Single	Bounded	❌
crossbeam::unbounded	Multi	Multi	Unbounded	✓
crossbeam::bounded	Multi	Multi	Bounded	✓
tokio::mpsc	Multi	Single	Bounded	✓*
tokio::broadcast	Multi	Multi	Bounded	✓*

* tokio канали для async коду

Рекомендації:
- Простий випадок → std::sync::mpsc
- MPMC або select! → crossbeam-channel
- Async код → tokio::sync

```
// oneshot — канал для одного повідомлення
// Ефективний для request-response pattern

// Можна реалізувати через sync_channel(1)
use std::sync::mpsc::sync_channel;

fn request_response() {
    let (response_tx, response_rx) = sync_channel(1);

    // Надсилаємо запит з каналом для відповіді
    request_tx.send(Request {
        data: "query",
        response_channel: response_tx,
    }).unwrap();

    // Чекаємо відповідь
    let response = response_rx.recv().unwrap();
}

// Або використовуйте tokio::sync::oneshot (async)
// або crossbeam_channel з bounded(1)

// futures::channel::oneshot для futures
use futures::channel::oneshot;
let (tx, rx) = oneshot::channel::<i32>();
tx.send(42).unwrap();
let result = rx.await;  // async
```

MAC Work Packages

```
use std::sync::mpsc::{sync_channel, Sender, Receiver};
use std::sync::{Arc, Mutex};
use std::thread;

type Job = Box<dyn FnOnce() + Send + 'static>;

struct ThreadPool {
    workers: Vec<thread::JoinHandle<()>>,
    sender: Sender<Job>,
}

impl ThreadPool {
    fn new(size: usize) -> Self {
        let (sender, receiver) = sync_channel::<Job>(size * 2);
        let receiver = Arc::new(Mutex::new(receiver));

        let workers = (0..size).map(|id| {
            let rx = Arc::clone(&receiver);
            thread::spawn(move || loop {
                let job = rx.lock().unwrap().recv();
                match job {
                    Ok(job) => job(),
                    Err(_) => break,  // Channel closed
                }
            })
        }).collect();

        ThreadPool { workers, sender }
    }
```

```rust
use std::sync::mpsc::{channel, sync_channel, Sender};

struct Request {
    query: String,
    response_tx: Sender<Response>,
}

struct Response {
    result: String,
}

// Сервіс обробки запитів
fn service(rx: Receiver<Request>) {
    for request in rx {
        let result = process_query(&request.query);
        request.response_tx.send(Response { result }).unwrap();
    }
}

// Клієнт
fn client(service_tx: &Sender<Request>) -> Response {
    let (response_tx, response_rx) = sync_channel(1);

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

    response_rx.recv().unwrap()  // Чекаємо відповідь
}
```

```rust
use crossbeam_channel::{unbounded, Sender, Receiver};
use std::thread;

#[derive(Clone, Debug)]
enum SwarmEvent {
    AgentJoined(u32),
    AgentLeft(u32),
    TargetDetected { pos: (f64, f64), by: u32 },
    MissionAssigned { mission_id: u32, agents: Vec<u32> },
}

struct EventBus {
    sender: Sender<SwarmEvent>,
    receiver: Receiver<SwarmEvent>,
}

impl EventBus {
    fn new() -> Self {
        let (sender, receiver) = unbounded();
        EventBus { sender, receiver }
    }

    fn publisher(&self) -> Sender<SwarmEvent> {
        self.sender.clone()
    }

    fn subscriber(&self) -> Receiver<SwarmEvent> {
        self.receiver.clone()  // MPMC — кілька subscribers!
    }
}
```

Конвеєрна обробка

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

// Stage 1: Отримання даних з сенсорів
// Stage 2: Фільтрація шуму
// Stage 3: Розпізнавання об'єктів
// Stage 4: Прийняття рішень

fn create_pipeline() {
    let (raw_tx, raw_rx) = sync_channel::<SensorData>(100);
    let (filtered_tx, filtered_rx) = sync_channel::<FilteredData>(50);
    let (detected_tx, detected_rx) = sync_channel::<Detection>(20);

    // Stage 2: Фільтрація
    thread::spawn(move || {
        for data in raw_rx {
            if let Some(filtered) = filter_noise(data) {
                filtered_tx.send(filtered).unwrap();
            }
        }
    });

    // Stage 3: Розпізнавання
    thread::spawn(move || {
        for data in filtered_rx {
            if let Some(detection) = detect_objects(data) {
                detected_tx.send(detection).unwrap();
            }
        }
    });
```

```rust
use crossbeam_channel::{bounded, Sender, Receiver};
use std::thread;

/// Fan-out: один producer → багато consumers
fn fan_out<T: Clone + Send + 'static>(
    source: Receiver<T>,
    n_workers: usize,
) -> Vec<Receiver<T>> {
    (0..n_workers).map(|_| {
        let (tx, rx) = bounded(10);
        let source = source.clone();  // MPMC!
        thread::spawn(move || {
            for item in source {
                // Кожен worker отримує частину роботи
                tx.send(item).ok();
            }
        });
        rx
    }).collect()
}

/// Fan-in: багато producers → один consumer
fn fan_in<T: Send + 'static>(
    sources: Vec<Receiver<T>>,
) -> Receiver<T> {
    let (tx, rx) = bounded(sources.len() * 10);
    for source in sources {
        let tx = tx.clone();
        thread::spawn(move || {
            for item in source {
```

```rust
use crossbeam_channel::{bounded, select, Sender, Receiver};

struct Coordinator {
    from_agents: Receiver<AgentMessage>,
    to_agents: Vec<Sender<Command>>,
    shutdown: Receiver<()>,
}

impl Coordinator {
    fn run(&mut self) {
        loop {
            select! {
                recv(self.from_agents) -> msg => {
                    if let Ok(msg) = msg {
                        self.handle_agent_message(msg);
                    }
                }
                recv(self.shutdown) -> _ => {
                    println!("Coordinator shutting down");
                    // Надіслати команду зупинки всім агентам
                    for tx in &self.to_agents {
                        let _ = tx.send(Command::Shutdown);
                    }
                    break;
                }
            }
        }
    }

    fn broadcast_command(&self, cmd: Command) {
```

Channel Patterns Summary

1. Producer-Consumer
   N producers → [channel] → 1 consumer

2. Pipeline
   Stage1 → [ch] → Stage2 → [ch] → Stage3

3. Fan-out / Fan-in
   1 → [ch] → N workers → [ch] → 1

4. Request-Response
   Client → [request + response_ch] → Server
   Client ← [response_ch] ← Server

5. Pub-Sub (broadcast)
   Publisher → [ch] → N subscribers

6. Worker Pool
   [job queue] → N workers (competing)

```
// Замість великих даних — передавайте Arc
let large_data = Arc::new(vec![0u8; 1_000_000]);
tx.send(Arc::clone(&large_data)).unwrap();  // Дешево!
```

Коли channels, коли mutex

Channels краще для:

- Передачі ownership
- Pipeline обробки
- Request-response
- Loose coupling
- Distributed systems
- Actor model
- Event-driven архітектури

Mutex/RwLock краще для:

- Shared mutable state
- Часті read/write
- Малі критичні секції
- Cache
- Counters
- Simple shared data

Часто комбінація: channels для commands, mutex для shared state

Best Practices

✓ Використовуйте bounded channels з розумним розміром
✓ Завжди drop() sender коли закінчили
✓ Handle errors від send/recv (не просто unwrap)
✓ Використовуйте select! для кількох каналів
✓ Типізуйте повідомлення через enum

❌ Не ігноруйте backpressure
❌ Не створюйте cyclic channel dependencies (deadlock)
❌ Не забувайте про graceful shutdown

🤖 Для MAC:
• Coordinator з select! на всіх каналах
• Typed messages: enum AgentMessage
• Response channels для request-response
• Bounded buffers для sensor data

```
use crossbeam_channel::{bounded, select, Receiver, Sender};
use std::thread;

struct Worker {
    work_rx: Receiver<Job>,
    shutdown_rx: Receiver<()>,
}

impl Worker {
    fn run(&self) {
        loop {
            select! {
                recv(self.work_rx) -> job => {
                    if let Ok(job) = job {
                        job.execute();
                    } else {
                        // Channel closed
                        break;
                    }
                }
                recv(self.shutdown_rx) -> _ => {
                    println!("Shutdown signal received");
                    // Finish current work, cleanup
                    break;
                }
            }
        }
        println!("Worker stopped gracefully");
    }
}
```

Підсумок лекції

sync_channel — bounded з backpressure:
• send() блокує коли буфер повний
• sync_channel(0) — rendezvous

crossbeam-channel:
• MPMC — multi-consumer!
• select! — очікування на кількох каналах
• Швидший за std

Patterns:
• Pipeline, Fan-out/Fan-in
• Request-Response з oneshot
• Worker Pool, Event Bus

🤖 MAC: channels для координації, mutex для shared state

→ Наступна лекція: Async/Await — асинхронне програмування

Завдання для самостійної роботи

1. Producer-Consumer:
   - 3 producers генерують числа
   - 1 consumer обчислює суму
   - Bounded buffer (10)

2. Pipeline:
   - Stage 1: генерація чисел
   - Stage 2: фільтрація парних
   - Stage 3: множення на 2
   - Stage 4: вивід

3. Worker Pool:
   - 4 workers
   - Обробка 100 задач
   - Graceful shutdown

4. MAC Coordinator:
   - 5 агентів надсилають позиції
   - Coordinator з select!
   - Broadcast команд
   - Shutdown signal

5. Request-Response:
   - Database service
   - Clients надсилають queries

Дякую за увагу!

Channels • mpsc • crossbeam • select!

Питання?