Ejemplos de Actores y Canales

Actix trabaja con un bloque async, porque trabaja de forma asincrónica.

1. helloworld.rs

Declaración de dependencias

```
extern crate actix;

use actix::{Actor, Context, Handler, System, Message};
```

Se define el mensaje SayHello que tiene un name (string) y cuando responde, entrega un string también.

```
//#[derive(Message)]
//#[rtype(result = "String")]
struct SayHello {
    name: String
}

impl Message for SayHello {
    type Result = String;
}
```

Greeter maneja el SayHello: cuando recibe, devuelve “Hello” + nombre.

```
struct Greeter {

}

impl Actor for Greeter {
    type Context = Context<Self>;
}

impl Handler<SayHello> for Greeter {
    type Result = String;

fn handle(&mut self, msg: SayHello, _ctx: &mut Context<Self>) ->
```

```
Self::Result {
        "Hello ".to_owned() + &msg.name
    }
}
```

En el main, se crea un actor Greeter (un saludador).

```
#[actix_rt::main]
async fn main() {
    let addr = Greeter {}.start(); // <- start actor and get its address
```

Se le manda un SayHello con el nombre como “World”. El await espera la respuesta y la imprime.

```
    // send message and get future for result
    let res = addr.send(SayHello { name: String::from("world!")
}).await;

    println!("{}", res.unwrap());
    System::current().stop();
}
```

2. estadointerno.rs

Es un actor con estado interno, llamado Calc que suma y resta.

Se define un actor Calc con un estado current (i32). Se definen dos tipos de mensajes: Sub(i32) y Add(i32).

```
#[derive(Message)]
#[rtype(result = "i32")]
struct Add(i32);

#[derive(Message)]
#[rtype(result = "i32")]
struct Sub(i32);

struct Calc {
    current: i32
}
```

```
impl Actor for Calc {
    type Context = Context<Self>;
}

impl Handler<Add> for Calc {
    type Result = i32;
```

Calc implementa:

- Handler<Add>: suma msg.0 (contenido del mensaje) al current.
- Handler<Sub>: resta msg.0 (contenido del mensaje) al current.

```
    fn handle(&mut self, msg: Add, _ctx: &mut Context<Self>) ->
Self::Result {
        println!("add {}", msg.0);
        self.current += msg.0;
        self.current
    }
}

impl Handler<Sub> for Calc {
    type Result = i32;

    fn handle(&mut self, msg: Sub, _ctx: &mut Context<Self>) ->
Self::Result {
        println!("sub {}", msg.0);
        self.current -= msg.0;
        self.current
    }
}
```

En main se crea un actor Calc y se envían mensajes.

```
#[actix_rt::main]
async fn main() {
    let addr = Calc { current: 0 }.start(); // <- start actor and get
its address
```

Lo manda y no espera.

```
    // fire and forget
    addr.do_send(Add(20));
```

```
    println!("do_send done");
```

Lo manda y si no puede, da error.

```
    // fire and forget, but check for errors like full mailbox
    addr.try_send(Add(15)).unwrap();
    println!("try_send done");
```

Espera y muestra el resultado. send me devuelve un Future con el mensaje ya procesado (pasado por el Handler).

```
   // wait for response
    let res = addr.send(Add(5)).await;
    println!("{}", res.unwrap());
```

Espera y muestra el resultado.

```
    // wait for response
    let res = addr.send(Sub(3)).await;

    println!("{}", res.unwrap());
    System::current().stop();
}
```

3. sleep.rs

El actor duerme cuando recibe un mensaje.

Define el mensaje Sleep.

```
#[derive(Message)]
#[rtype(result = "()")]
struct Sleep(u64);

struct Sleepyhead {
    id: usize
}
```

El actor Sleepyhead maneja Sleep.

```
impl Actor for Sleepyhead {
    type Context = Context<Self>; //SyncContext<Self>;
}
```

Cada vez que recibe hace `tokio::time::sleep()` durante la cantidad de segundos que indica el mensaje.

```
#[async_handler]
impl Handler<Sleep> for Sleepyhead {
    type Result = ();

    async fn handle(&mut self, msg: Sleep, _ctx: &mut <Sleepyhead as
Actor>::Context) -> Self::Result  {
        println!("[{}] durmiendo por {}", self.id, msg.0);
        tokio::time::sleep(Duration::from_secs(msg.0)).await;
        println!("[{}] desperté de {}", self.id, msg.0);
    }
}
```

En main se crean 2 Sleepyhead.

```
#[actix_rt::main]
async fn main() {
    // console_subscriber::init();

    println!("Enter para empezar");
    io::stdin().read(&mut [0u8]).unwrap();

    let addr = Sleepyhead { id: 1}.start(); // <- start actor and get
its address
    // let addr = SyncArbiter::start(1, || Sleepyhead { id: 1 });
    // let addr = SyncArbiter::start(2, || Sleepyhead { id: 1 });

    let other = Sleepyhead { id: 2 }.start(); // <- start actor and get
its address
    // let other = SyncArbiter::start(1, || Sleepyhead { id: 2 });
```

Se envía “3” al actor 1.
Se envía “2” al actor 2.

```
    let now = SystemTime::now();
```

```
    addr.try_send(Sleep(3)).unwrap();
    println!("mandé 3 al 1");

    other.try_send(Sleep(2)).unwrap();
    println!("mandé 2 al 2");
```

Se envía mensaje y se espera la respuesta.

```
    // wait for response
    addr.send(Sleep(2)).await.unwrap();
```

```
    println!("terminé. tardé {}", now.elapsed().unwrap().as_secs());


    println!("Enter para terminar");
    io::stdin().read(&mut [0u8]).unwrap();

    System::current().stop();
}
```

4. banquero.rs

Solución con actores del problema del banquero.

Defino 5 inversores.

```
const INVERSORES:usize = 5;
```

Mensaje Invertir{amount, sender}: mensaje de inversión enviado a un inversor.

```
#[derive(Message)]
#[rtype(result = "()")]
struct Invertir {
    amount: f64,
    sender: Recipient<ResultadoInversion>
}
```

Respuesta del inversor.

```rust
#[derive(Message)]
#[rtype(result = "()")]
struct ResultadoInversion(usize, f64);
```

Inicia una semana con cierta cantidad de dinero.

```rust
#[derive(Message, Debug)]
#[rtype(result = "()")]
struct Semana(f64);
```

Un banquero tiene el dinero, una lista de inversores y un hash de devoluciones (qué inversores ya devolvieron el dinero de su inversión esta semana).

```rust
struct Banquero {
    plata: f64,
    inversores: Vec<Recipient<Invertir>>,
    devoluciones: HashSet<usize>
}
```

Un inversor tiene un ID.

```rust
struct Inversor {
    id: usize,
}
```

```rust
impl Actor for Banquero {
    type Context = Context<Self>;
}

impl Actor for Inversor {
    type Context = Context<Self>;
}
```

El actor Banquero recibe el mensaje Semana.

```rust
impl Handler<Semana> for Banquero {
    type Result = ();
```

Divide equitativamente el dinero entre los inversores.

```
    fn handle(&mut self, _msg: Semana, _ctx: &mut Context<Self>) ->
Self::Result {
        let plata = _msg.0;
        let amount = plata / self.inversores.len() as f64;
        self.plata = 0.;
        self.devoluciones.clear();

        println!("[BANQUERO] empieza la semana con {}", plata);
```

Les manda a todos los inversores el mensaje Invertir, con el monto de dinero que le toca y su dirección para que luego puedan enviarle la ganancia.

```
        for inversor in self.inversores.iter() {
            inversor.try_send(Invertir { amount, sender:
_ctx.address().recipient()}).unwrap();
        }


    }
}
```

Banquero recibe ResultadoInversion.

```
impl Handler<ResultadoInversion> for Banquero {
    type Result = ();
```

Suma el dinero recibido, se fija si el inversor ya lo devolvió, si no lo hizo, suma su dinero al total y lo ingresa a la lista de devoluciones. Cuando los que ya devolvieron es igual a la cantidad de inversores, muestra el resultado final y comienza una nueva semana.

```
    fn handle(&mut self, msg: ResultadoInversion, _ctx: &mut
Context<Self>) -> Self::Result {

        println!("[BANQUERO] recibí resultado de la inversion {}",
msg.0);

        if !self.devoluciones.contains(&msg.0) {
```

```
            self.plata += msg.1;
            self.devoluciones.insert(msg.0);

            if self.devoluciones.len() == self.inversores.len() {
                println!("[BANQUERO] fin de la semana, resultado final
{}", self.plata);
                _ctx.address().try_send(Semana(self.plata)).unwrap();
            }
        }

    }
}
```

Inversor recibe Invertir.

```
#[async_handler]
impl Handler<Invertir> for Inversor {
```

Simula trabajar en la inversión con un sleep, genera un resultado, devuelve a través de ResultadoInversión lo que generó.

```
    type Result = ();

    fn handle(&mut self, msg: Invertir, _ctx: &mut Context<Self>) ->
Self::Result {
        println!("[INV {}] recibo inversion por {}", self.id,
msg.amount);
        sleep(Duration::from_millis(thread_rng().gen_range(500,
1500))).await;
        let resultado = msg.amount * thread_rng().gen_range(0.5, 1.5);
        println!("[INV {}] devuelvo {}", self.id, resultado);
        msg.sender.try_send(ResultadoInversion(self.id,
resultado)).unwrap();
    }
}
```

```
fn main() {
    let system = System::new();
    system.block_on(async {
        let mut inversores = vec!();

        for id in 0..INVERSORES {
            inversores.push(Inversor { id }.start().recipient())
```

```
        }


        Banquero { plata: 0.0, inversores, devoluciones:
HashSet::with_capacity(INVERSORES) }.start()
            .do_send(Semana(1000.0));
    });

    system.run().unwrap();
}
```

5. banquero_channel.rs

Defino 10 inversores.

```
const INVERSORES: i32 = 10;
```

Se crea un channel devolucion_send / devolucion_receive.

```
fn main() {
    let mut plata = 1000.0;

    let (devolucion_send, devolucion_receive) = mpsc::channel();
```

Cada inversor es un thread.
Cada semana:

- Se divide el dinero.
- Se manda a cada inversor cuánto le toca.
- Espera todas las devoluciones
- Imprime el resultado
- Vuelve a empezar.

```
    let inversores: Vec<(Sender<f64>, JoinHandle<()>)> = (0..INVERSORES)
        .map(|id| {
            let (inversor_send, inversor_receive) = mpsc::channel();
            let devolucion_send_inversor = devolucion_send.clone();
            let t = thread::spawn(move || inversor(id, inversor_receive,
devolucion_send_inversor));
            (inversor_send, t)
        })
        .collect();
```

```
       loop {
        let mut plata_semana = iniciar_semana(&mut plata, &inversores);

        let mut devolvieron = HashSet::new();

        while(devolvieron.len() < (INVERSORES as usize)) {
            let (who, how_much) = devolucion_receive.recv().unwrap();
            println!("[Banquero] recibí de {} el monto {}", who,
how_much);
            if !devolvieron.contains(&who) {
                devolvieron.insert(who);
                plata_semana += how_much;
            }
        }

        println!("[Banquero] final de semana {}", plata_semana);
        plata = plata_semana
    }

    let _:Vec<()> = inversores.into_iter()
        .flat_map(|(_,h)| h.join())
        .collect();
}
```

Divide el dinero para cada inversor y lo envía por el canal del inversor.

```
fn iniciar_semana(plata: &mut f64, inversores: &Vec<(Sender<f64>,
JoinHandle<()>)>) -> f64 {
    let prestamo = *plata / (INVERSORES as f64);
    for (inversor, _) in inversores {
        inversor.send(prestamo).unwrap();
    }

    let mut plata_semana = 0.0;
    plata_semana
}
```

Cada inversor espera recibir qué le mandaron, hace un sleep para simular, calcula el resultado y se lo envía al banquero por devolucion_send.

```
fn inversor(id: i32, prestamo: Receiver<f64>, devolucion: Sender<(i32,
```

```
f64)>) {
    loop {
        let plata_inicial = prestamo.recv().unwrap();
        println!("[Inversor {}] me dan {}", id, plata_inicial);
        thread::sleep(Duration::from_secs(2));
        let resultado = plata_inicial * thread_rng().gen_range(0.5,
1.5);
        println!("[Inversor {}] devuelvo {}", id, resultado);
        devolucion.send((id, resultado));
    }
}
```

6. filosofos.rs

Defino 5 filósofos.

```
const N: usize = 5;
```

Hashmap que representa “este palito está compartido con este vecino”.

```
type Neighbours = HashMap<ChopstickId, Addr<Philosopher>>;
```

Cada filósofo va a manejar mensajes que le llegan:
Think, Hungry, TryToEat, EatingDone, ChopstickRequest, ChopstickResponse, SetNeighbours.

```
#[derive(Message)]
#[rtype(result = "()")]
struct SetNeighbours(Neighbours);

#[derive(Message)]
#[rtype(result = "()")]
struct Think;

#[derive(Message)]
#[rtype(result = "()")]
struct Hungry;

#[derive(Message)]
#[rtype(result = "()")]
```

```rust
struct ChopstickRequest(ChopstickId);

#[derive(Message)]
#[rtype(result = "()")]
struct ChopstickResponse(ChopstickId);

#[derive(Message)]
#[rtype(result = "()")]
struct TryToEat;

#[derive(Message)]
#[rtype(result = "()")]
struct EatingDone;
```

Estados del palito.

```rust
#[derive(PartialEq)]
enum ChopstickState {
    DontHave,
    Dirty,
    Clean,
    Requested
}
```

Cada palito tiene su ID.

```rust
#[derive(PartialEq, Eq, Hash, Copy, Clone)]
struct ChopstickId(usize);
```

Un filósofo guarda su id, qué palitos tiene y a quién puede pedirle palitos.

```rust
struct Philosopher {
    id: usize,
    chopsticks: HashMap<ChopstickId, ChopstickState>,
    neighbours: Neighbours
}

impl Actor for Philosopher {
    type Context = Context<Self>;
}
```

Envío de los vecinos que tiene el filósofo.

```
impl Handler<SetNeighbours> for Philosopher {
    type Result = ();

    fn handle(&mut self, msg: SetNeighbours, ctx: &mut Context<Self>) ->
Self::Result {
        println!("[{}] recibi a mis vecinos", self.id);
        self.neighbours = msg.0;
        ctx.address().try_send(Think).unwrap();
    }
}
```

Cuando el filósofo recibe Think, piensa un tiempo entre 2 y 5 segundos, luego se pone hambriento y se manda a sí mismo el mensaje Hungry.

```
#[async_handler]
impl Handler<Think> for Philosopher {
    type Result = ();

    async fn handle(&mut self, msg: Think, ctx: &mut Context<Self>) ->
Self::Result {
        println!("[{}] pensando", self.id);
        sleep(Duration::from_millis(thread_rng().gen_range(2000,
5000))).await;
        ctx.address().try_send(Hungry).unwrap();
    }
}
```

Cuando el filósofo recibe TryToEat: revisa si tiene todos los palitos, si los tiene, come, sino no come. Si puede comer, manda EatingDone.

```
#[async_handler]
impl Handler<TryToEat> for Philosopher {
    type Result = ();

    async fn handle(&mut self, msg: TryToEat, ctx: &mut Context<Self>)
-> Self::Result {
        if self.chopsticks.iter().all(|(_id, state)| *state !=
ChopstickState::DontHave) { // si los tengo todos
            println!("[{}] comiendo", self.id);
```

```
            sleep(Duration::from_millis(thread_rng().gen_range(2000,
5000))).await;
            ctx.address().try_send(EatingDone).unwrap();
        } else {
            println!("[{}] aun no puedo comer", self.id);
        }
    }
}
```

Cuando el filósofo recibe Hungry mira qué palitos no tiene, pide esos palitos a sus vecinos enviando ChopstickRequest y después se manda a sí mismo TryToEat.

```
#[async_handler]
impl Handler<Hungry> for Philosopher {
    type Result = ();

    async fn handle(&mut self, _msg: Hungry, ctx: &mut Context<Self>) ->
Self::Result {
        println!("[{}] por comer", self.id);
        for (chopstick_id, state) in self.chopsticks.iter() {
            if *state == ChopstickState::DontHave {
                println!("[{}] pido palito {}", self.id,
chopstick_id.0);

self.neighbours.get(chopstick_id).unwrap().try_send(ChopstickRequest(*ch
opstick_id)).unwrap();
            }
        }

        ctx.address().try_send(TryToEat).unwrap();
    }
}
```

Cuando un filósofo recibe ChopstickRequest: revisa si tiene el palito, si está sucio, se lo da (osea ese palito ya lo usó) y marca que ya no lo tiene. si está limpio, no se lo da todavía, lo marca como Requested.

Protocolo Chandy/Misra:

- Si está sucio, lo da rápido.
- Si está limpio, lo retiene hasta que termine de comer.

```rust
impl Handler<ChopstickRequest> for Philosopher {
    type Result = ();

    fn handle(&mut self, msg: ChopstickRequest, _ctx: &mut
Context<Self>) -> Self::Result {
        println!("[{}] me piden palito {}", self.id, msg.0.0);
        let chopstick = msg.0;
        let chopstick_state = &self.chopsticks.get(&chopstick);
        match chopstick_state {
            Some(ChopstickState::Dirty) => {
                println!("[{}] se lo doy ahora", self.id);

self.neighbours.get(&chopstick).unwrap().try_send(ChopstickResponse(msg.
0)).unwrap();
                self.chopsticks.insert(chopstick,
ChopstickState::DontHave);
            },
            Some(ChopstickState::Clean) => {
                println!("[{}] se lo doy cuando termine", self.id);
                self.chopsticks.insert(chopstick,
ChopstickState::Requested);
            },
            _ => {
                println!("[{}] no deberia pasar", self.id);
            }
        }
    }
}
```

Cuando un filósofo recibe ChopstickResponse:

- Actualiza su estado (pasa a Clean)
- Vuelve a intentar comer con TryToEat

```rust
#[async_handler]
impl Handler<ChopstickResponse> for Philosopher {
    type Result = ();

    async fn handle(&mut self, msg: ChopstickResponse, ctx: &mut
Context<Self>) -> Self::Result {
        println!("[{}] recibi palito {}", self.id, msg.0.0);
        self.chopsticks.insert(msg.0,ChopstickState::Clean);
        ctx.address().try_send(TryToEat).unwrap();
    }
}
```

Cuando un filósofo recibe EatingDone:

- Para cada palito que tiene: si alguien lo había pedido, se lo da y marca que ya no lo tiene. Si nadie lo pidió, simplemente lo ensucia.

Después vuelve a mandar Think para que empiece el ciclo otra vez.

```
#[async_handler]
impl Handler<EatingDone> for Philosopher {
    type Result = ();

    async fn handle(&mut self, _msg: EatingDone, ctx: &mut
Context<Self>) -> Self::Result {
        println!("[{}] terminé de comer", self.id);
        for (chopstick, mut state) in self.chopsticks.iter_mut() {
            if *state == ChopstickState::Requested {
                println!("[{}] entrego palito {}", self.id,
chopstick.0);

self.neighbours.get(chopstick).unwrap().try_send(ChopstickResponse(*chop
stick)).unwrap();
                *state = ChopstickState::DontHave
            } else {
                println!("[{}] marco como sucio palito {}", self.id,
chopstick.0);
                *state = ChopstickState::Dirty
            }
        }
        ctx.address().try_send(Think).unwrap();
    }
}
```

Inicialización en main:

- Creación de cada filósofo
- Seteo de los palitos que tiene cada filósofo:
  - Cada filósofo tiene 2 palitos relacionados (palito de su ID Y el de su vecino derecho).
  - El filósofo 0, su palito propio empieza como Dirty y el filósofo N-1, su palito del vecino (0) empieza como DontHave.
  - Los demás tienen sus palitos de forma alternada: uno Dirty y el otro DontHave.

Todo esto es para evitar deadlocks en el inicio, además evitamos

ciclos.

```
fn main() {
    let system = System::new();
    system.block_on(async {
        let mut philosophers = vec!();

        for id in 0..N {
            // Deadlock avoidance forcing the initial state
            philosophers.push(Philosopher {
                id,
                chopsticks: HashMap::from([
                    (ChopstickId(id), if id == 0 { ChopstickState::Dirty
} else { ChopstickState::DontHave }),
                    (ChopstickId((id + 1) % N), if id == N-1 {
ChopstickState::DontHave } else { ChopstickState::Dirty })
                ]),
                neighbours: HashMap::with_capacity(2)
            }.start())
        }


        for id in 0..N {
            let prev = if id == 0 { N - 1 } else { id - 1 };
            let next = (id + 1) % N;
            philosophers[id].try_send(SetNeighbours(HashMap::from([
                (ChopstickId(id), philosophers[prev].clone()),
                (ChopstickId(next), philosophers[next].clone())
            ]))).unwrap();
        }
    });

    system.run().unwrap();
}
```