Trabajo Teoría:
Rust async / await

Programación Concurrente

Francisco Javier Pérez Martínez

11 de diciembre de 2021

Índice

1. Introducción: ¿Qué es Rust?

Rust es un lenguaje de programación *compilado* (como C, C++, Java) y *multiparadigma* desarrollado por Mozilla. Ha sido elaborado para ser un **lenguaje seguro, concurrente y práctico**. Además, soporta programación funcional pura, imperativa, por procedimientos y orientada a objetos.

Rust sigue una política en la que busca la opinión y contribución de los miembros de su comunidad. Su diseño se ha ido perfeccionando a través de las vivencias en el desarrollo del motor del navegador Servo (nuevo motor renderizado libre escrito en Rust) y por su propio compilador.

Actualmente, es uno de los lenguajes de programación más utilizados en cuanto al uso de criptomonedas y la creación de nodos para minar activos.

Figura 1: Rust logo

1.1. Descripción

Además de los objetivos mencionados anteriormente, el propósito de Rust es ser un buen lenguaje para la creación de grandes programas del tipo cliente/servidor ejecutados en Internet, por lo que han enfatizado en características de seguridad, control de distribución de la memoria y una concurrencia segura. El sistema está diseñado para tener un acceso seguro a la memoria y no permite punteros nulos o colgantes.

El sistema de tipos soporta un mecanismo parecido a las clases de tipos, llamado **"traits"** facilitando el polimorfismo con varios tipos de argumentos (polimorfismo ad-hoc), el cual ha sido logrado mediante la adición de restricciones para escribir declaraciones de variables.

Los traits no son más que una lista de métodos con o sin implementación que las estructuras o los tipos deben de implementar para cumplir ese trait.

Rust cuenta con **inferencia de tipos** (como Vala C#, etc), para las variables declaradas con el atributo let. No es necesario que sean inicializadas con un valor por defecto.

Existen dos formas de escribir codigo en Rust: **Safe Rust y Unsafe Rust**. La diferencia está en que el "modo seguro" le impone restricciones adiciones al programador garantizando que el código funcione correctamente y el "modo no seguro" aunque le proporciona más autonomía al programador, el código puede romperse con mayor facilidad.

Este modelo dual de Rust es una de sus mayores ventajas, cosa que en C++ por ejemplo, nunca se sabe que se ha escrito un código inseguro hasta que en algún instante el programa falla o aparece una brecha de seguridad.

1.2. Concurrencia en Rust

Rust posee un sistema de tipos estático que nos facilita la programación concurrente con las comprobaciones en tiempo de compilación y con su gestión de memoria. Este sistema, nos asegura que no ocurrirán condiciones de carrera por lo que una vez compilado tendremos la certeza de que la gestión de la memoria compartida es correcta y no se producirá ningún fallo de segmentación.

Como tal, Rust nos provee de dos traits a la hora de trabajar con código concurrente:

- **Send**: este trait le indica al compilador que cualquier cosa de este tipo puede transferir la pertenencia entre hilos de forma segura.
- **Sync**: este trait le indica al compilador que cualquier cosa de este tipo no tiene posibilidad de introducir inseguridad en memoria cuando es usado de forma concurrente por múltiples hilos de ejecución.

Estos dos traits, nos permiten usar el sistema de tipos garantizando seguridad en las propiedades de nuestro código concurrente. Además, tenemos los tipos **Arc** y **Mutex** si queremos compartir una variable entre hilos. Y en cuanto a la comunicación entre diferentes hilos, la biblioteca estándar cuenta con **canales** para la sincronización entre hilos.

Además de usar los threads básicos de Rust, existen otras muchas bibliotecas que ofrecen otra manera de crear código concurrente:

- Crossbeam
- Rayon
- Coroutine-rs
- Futures

En la siguiente sección, nos centraremos en esta última biblioteca (Futures) la cual está enfocada en la programación asíncrona siendo esta el tema de nuestro documento.

2. Programación asíncrona en Rust

La programación asíncrona es un modelo de programación concurrente compatible con una gran número de lenguajes de programación. Esta tipo de programación, nos permite ejecutar una gran cantidad de tareas concurrentes en un pequeño número de hilos del sistema operativo, conservando una gran parte del aspecto de la programación síncrona ordinaria, a través de los atributos **async/await**.

2.1. Async frente a otros modelos de concurrencia

Para entender como encaja la programación asíncrona en el campo más amplio de la programación concurrente es necesario realizar una breve descripción de los modelos de concurrencia más populares:

- Los **hilos del SO** no requieren de ningún cambio en el modelo de programación, lo que hace que sea muy fácil de expresar la concurrencia. Sin embargo, la sincronización entre hilos puede ser complicada ya que pueden aparecer grandes sobrecargas de rendimiento. Los thread pools pueden mitigar algunos de estos costes, pero no lo suficiente como para soportar cargas de trabajo masivas de E/S.
- La **programación dirigida por eventos**, junto con *callbacks*, puede ser muy eficaz, pero tiende a dar un flujo de control verboso y "no lineal". El flujo de datos y la propagación de errores son frecuentemente difíciles de seguir.
- Las **corrutinas**, al igual que async, pueden admitir una gran cantidad de tareas. Sin embargo, abstraen los detalles de bajo nivel que son importantes para la programación de sistemas y los implementadores de tiempo de ejecución personalizados.
- El **modelo de actor** divide todo el cómputo concurrente en unidades llamadas actores, que se comunican a través del paso de mensajes como en los sistemas distribuidos. Este modelo, se puede implementar eficientemente, pero deja muchos problemas como el control de flujo y la lógica de reintento.

En resumen, la programación asíncrona nos permite implementaciones de alto rendimiento que son apropiados para lenguajes de bajo nivel como Rust, proporcionando la mayoría de los beneficios ergonómicos de los hilos y las corrutinas.

2.2. Async frente a otros lenguajes de programación

Muchos de los lenguajes actuales soportan la programación asíncrona, cada lenguaje con detalles de implementación propios de este mismo. La implementación de Rust difiere de la mayoría de los lenguajes en algunos aspectos:

- Los Futures son inertes en Rust y sólo avanzan cuando se les pregunta. Si un Future cae, deja de avanzar.
- Async es de coste cero en Rust, lo que significa que sólo se paga por lo que se utiliza. Específicamente, puedes usar async sin asignaciones en la pila y envío dinámico, lo que es genial para el rendimiento, el cual nos permite usarlo en sistemas embebidos.
- Los tiempos de ejecución son proporcionados por crates, unidades de compilación, mantenidos por la comunidad ya que Rust no proporciona ningún tiempo de ejecución integrado.

3. ¿Cómo funciona async / .await?

Los atributos incorporados de Rust para escribir funciones asíncronas que parecen código síncronos son **async / .await**. Async transforma un bloque de código en una máquina de estado que implementa un trait llamado **Future**.

Mientras que llamar a una función de bloqueo en un método síncrono bloquearía todo el hilo, los Futures bloqueados cederán el control del hilo, permitiendo la ejecución de otros Futures.

En primer lugar, debemos añadir algunas dependencias al archivo Cargo.toml:

```
[dependencies]
futures = "0.3"
```

Este archivo de configuración es el encargado de almacenar la información sobre el paquete, nombre, versión, información de autor y versión de Rust.

Para crear una función asíncrona, usaremos la siguiente sintaxis:

```
1    fn main() {
2        async fn do_something() { /* ... */ }
3    }
```

El valor que devuelve esta función asíncrona es un Future. Para que suceda algo, es necesario ejecutar dicho Future en un ejecutor.

3.1. Executor

Ejemplo de executor:

```rust
// `block_on` bloquea el hilo actual hasta que el Future proporcionado
// se haya ejecutado hasta el final.
// Otros ejecutores proporcionan un comportamiento más complejo,
// como programar múltiples futuros en el mismo hilo.
use futures::executor::block_on;

async fn hello_world() {
    println!("hello, world!");
}

fn main() {
    let future = hello_world(); // No se imprime nada
    block_on(future); // `Future` se ejecuta y "hello, world!" se imprime.
}
```

Dentro de una función asíncrona, podemos usar **.await** para esperar la finalización de otro tipo que implemente el trait Future, como la salida de otra función asíncrona.

A diferencia de **block_on**, .await no bloquea el hilo actual, sino que espera asincrónicamente a que se complete el Future, lo que permite que se ejecuten otras tareas si dicho trait no puede progresar actualmente.

Por ejemplo, imaginemos que tenemos 3 funciones asíncronas: **learn_song, sing_song,** y **dance**:

```rust
async fn learn_song() -> Song { /* ... */ }
async fn sing_song(song: Song) { /* ... */ }
async fn dance() { /* ... */ }
```

Una forma de hacer estas funciones sería bloquear cada una ellas individualmente utilizando **block_on**:

```rust
fn main() {
    let song = block_on(learn_song());
    block_on(sing_song(song));
    block_on(dance());
}
```

Sin embargo, de esta forma sólo hacemos una cosa la vez. Esta claro que tenemos que aprender la canción antes de poder cantarla, pero es posible bailar al mismo tiempo que aprendemos y cantamos la canción.

Para ello, podemos crear dos funciones asíncronas separadas que puedan ejecutarse simultáneamente:

```
async fn learn_and_sing() {
    // Espera a aprender la canción antes de cantarla.
    // Aquí utilizamos `.await` en lugar de `block_on` para evitar el bloqueo del hilo
    // que permite "bailar" al mismo tiempo.
    let song = learn_song().await;
    sing_song(song).await;
}

async fn async_main() {
    let f1 = learn_and_sing();
    let f2 = dance();

    // `join!` es como `.await` pero puede esperar varios Futures simultáneamente.
    // Si estamos temporalmente bloqueados en el Future de `learn_and_sing`, el Future `dance`
    // se hará cargo del hilo actual. Si `dance` se bloquea,
    // `learn_and_sing` puede recuperar el control. Si ambos Futures están bloqueados, entonces
    // `async_main` se bloquea y cede al ejecutor
    futures::join!(f1, f2);
}

fn main() {
    block_on(async_main());
}
```

En este ejemplo, el aprendizaje de la canción debe ocurrir antes de cantar la canción, pero tanto el aprendizaje como el canto pueden ocurrir al mismo tiempo que el baile. Si utilizáramos **block_on(learn_song())** en lugar de **learn_song().await** en **learn_and_sing**, el hilo no podría hacer nada más mientras **learn_song** se estuviera ejecutando. Esto haría imposible bailar al mismo tiempo. Al hacer **.await** en el Future de **learn_song**, permitimos que otras tareas tomen el control del hilo actual si **learn_song** se bloquea. Esto hace posible ejecutar múltiples Futures hasta su finalización de forma concurrente en el mismo hilo.

Como hemos visto hasta ahora, las funciones de async / .await hacen posible ceder el control del hilo actual en lugar de bloquearlo, lo que permite que otro código progrese mientras se espera que se completa una operación.

Hay 2 formas principales de utilizar async: mediante **funciones asíncronas (async fn)** y **bloques asíncronos.** Ambos devuelven un valor que implementan el rasgo **Future**.

```
// `foo()` devuelve un tipo que implementa `Future<Output = u8>`.
// `foo().await` dará como resultando un valor de tipo `u8`.
async fn foo() -> u8 { 5 }

fn bar() -> impl Future<Output = u8> {
    // Este bloque asíncrono da como resultando un tipo que implementa
    // `Future<Output = u8>`.
    async {
        let x: u8 = foo().await;
        x + 5
    }
}
```

3.2. El trait Future

El trait Future representa una computación asíncrona, es un valor que puede no haber terminado de computar todavía. Este tipo de "valor asíncrono" hace posible que un hilo continúe ejecutando código mientras espera que el valor esté disponible.

```
1  pub trait Future {
2      type Output;
3      fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output>;
4  }
5
6  enum Poll<T> {
7      Ready(T),
8      Pending,
9  }
```

Como podemos observar, tenemos dos componentes principales:

- El tipo asociado: **type Output**, que es el valor que devuelve el trait si existe.
- La función **fn poll**: que contiene 2 parámetros, Self y Context y nos devuelve un tipo Enum llamado Poll para saber si el valor está listo(Ready(T)) o se encuentra pendiente(Pending).
  - El primer parámetro: nos permite crear Futures que son inamovibles (constantes) usando **Pin** ya que es necesario para habilitar las funciones de async/await.
  - El segundo parámetro: se utiliza el tipo Context para tener acceso a un valor de tipo Waker para así saber que tarea debe ejecutarse específicamente.

Cuando un Future no está listo todavía, la función poll devuelve **Poll::Pending** y almacena un copia del Waker copiada sobre el **Context** actual (el tipo **Waker** se usa para decirle al ejecutor que la tarea asociada debe ejecutarse mediante el uso del método wake()).

Este Waker se despierta una vez que el Future puede avanzar e intenta evaluar (poll) el Future otra vez, lo que puede o no obtener un valor final.

3.3. Ejecución de múltiples Futures a la vez

Hasta el momento, hemos visto ejemplos principalmente de futures mediante el uso de .await, que bloquea la tarea actual hasta que un Future en concreto se completa. Sin embargo, las aplicaciones asíncronas reales a menudo necesitan ejecutar varias operaciones diferentes al mismo tiempo.

3.3.1. Join!

join! es una macro que permite esperar a que se completen varios Futures diferentes mientras se ejecutan todos al mismo tiempo.

Retomando el ejemplo de las 3 funciones asíncronas usado en el apartado 3.1 se puede ver como se usa la macro join! para esperar los Futures f1 y f2 simultáneamente. Si no lo hiciéramos de esta forma, el Future f2 no comenzaría a ejecutarse hasta que el Future f1 se haya completado. En otros lenguajes, los futures se ejecutan hasta su finalización, por lo que se pueden ejecutar dos operaciones al mismo tiempo llamando primero a cada función asíncrona para iniciar los futures y luego esperar a ambas.

Sin embargo, los Futures en Rust no harán nada hasta que se activen usando .await. Esto significa que si sólo usamos .await, dichos futures se ejecutarán en serie en lugar de ejecutarlos al mismo tiempo. Para ejecutar correctamente los dos futures al mismo tiempo utilizamos la macro **futures::join!**.

El valor devuelto por join! es una tupla que contiene la salida de cada Future pasado como parámetro.

3.3.2. Select!

select! es una macro que ejecuta múltiples Futures simultánemaente, lo que permite al usuario responder tan pronto como se complete cualquier future.

```rust
#![allow(unused)]
fn main() {
    use futures::{
        future::FutureExt, // para `.fuse()`
        pin_mut,
        select,
    };
    async fn task_one() { /* ... */ }
    async fn task_two() { /* ... */ }

    async fn race_tasks() {
        let t1 = task_one().fuse();
        let t2 = task_two().fuse();

        pin_mut!(t1, t2);

        select! {
            () = t1 => println!("task one completed first"),
            () = t2 => println!("task two completed first"),
        }
    }
}
```

La función anterior ejecutará tanto t1 y t2 al mismo tiempo. Cuando finalice t1 o t2, el controlador correspondiente llamará a println! y la función finalizará sin completar la tarea restante.

La sintaxis básica de select es <pattern> = <expression> ⇒ <*code*>.

4. Async Runtimes

Async runtimes o tiempos de ejecución asíncronos son bibliotecas utilizadas para ejecutar aplicaciones asíncronas. Suelen agrupar un **reactor** (bucle de eventos que controla todos los recursos de E/S) con uno o más **ejecutores**. Los reactores proporcionan mecanismos de suscripción para eventos externos, como E/S asíncronas, comunicación entre procesos y temporizadores. Esto es conocido como el patrón Reactor-Ejecutor.

En un tiempo de ejecución asíncrono, los suscriptores suelen ser futuros que representan operaciones de E/S de bajo nivel. Los ejecutores manejan la planificación y ejecución de tareas. Además, realizan un seguimiento de las tareas suspendidas y en ejecución, evalúan los futures hasta su finalización y activan las tareas cuando pueden avanzar.

A lo largo del documento hemos visto como los Futures poseen su propio ejecutor, pero no su reactor, ya que no admite la ejecución de E/S asíncronas. Por esta razón, no se considera un runtime completo. Una opción común es usar los Futures con un ejecutor de otro biblioteca.

4.1. Async Runtimes más populares

No hay un runtime asíncrono en la biblioteca estándar de Rust como tal sino que se necesitan de runtimes externos proporcionados por la comunidad. Las siguientes bibliotecas proporcionan los runtime más populares:

- **Tokio** : un ecosistema asíncrono popular con HTTP, gRPC y marcos de seguimiento.
- **async-std** : una biblioteca que proporciona contra-partes asíncronas a los componentes de la biblioteca estándar.
- **smol** : un runtime asíncrono pequeño y simplificado.
- **fuchsia-async** : un ejecutor para usar en Fuchsia OS, un sistema operativo de tiempo real que está siendo desarrollado por Google.

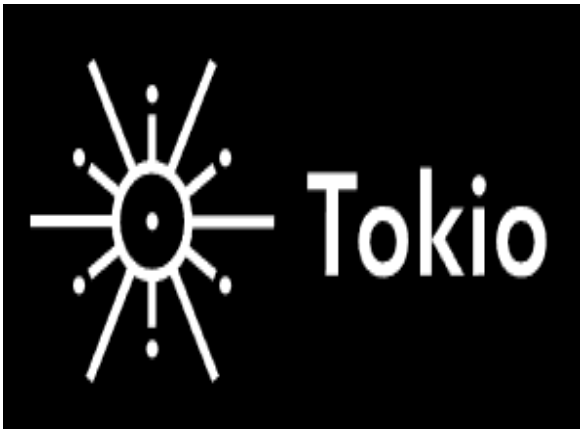

Figura 2: Tokio logo

Figura 3: Async-std logo

4.2. Tokio

Tokio es un runtime asíncrono para escribir aplicaciones en internet seguras sin comprometer la velocidad. Algunos de los componentes más importantes que proporciona son:

- Herramientas para trabajar con tareas asíncronas, incluidas primitivas y canales de sincronización y tiempos de espera, inactividad e intervalos.
- APIs para realizar operaciones de E/S asíncronas, incluidos sockets TCP y UDP, operaciones del sistema de archivos y gestión de procesos y señales.
- Un runtime para ejecutar código asíncrono, que incluye un programador de tareas, un controlador de E/S respaldado por la cola de eventos del sistema operativo (epoll, kqueue, IOCP, etc.) y un temporizador de alto rendimiento. Estos ejemplos mencionados, son métodos del kernel que el runtime necesita entender para realizar operaciones de E/S que el paquete (crate) **mio** implementa el cual se encuentra internamente en Tokio.

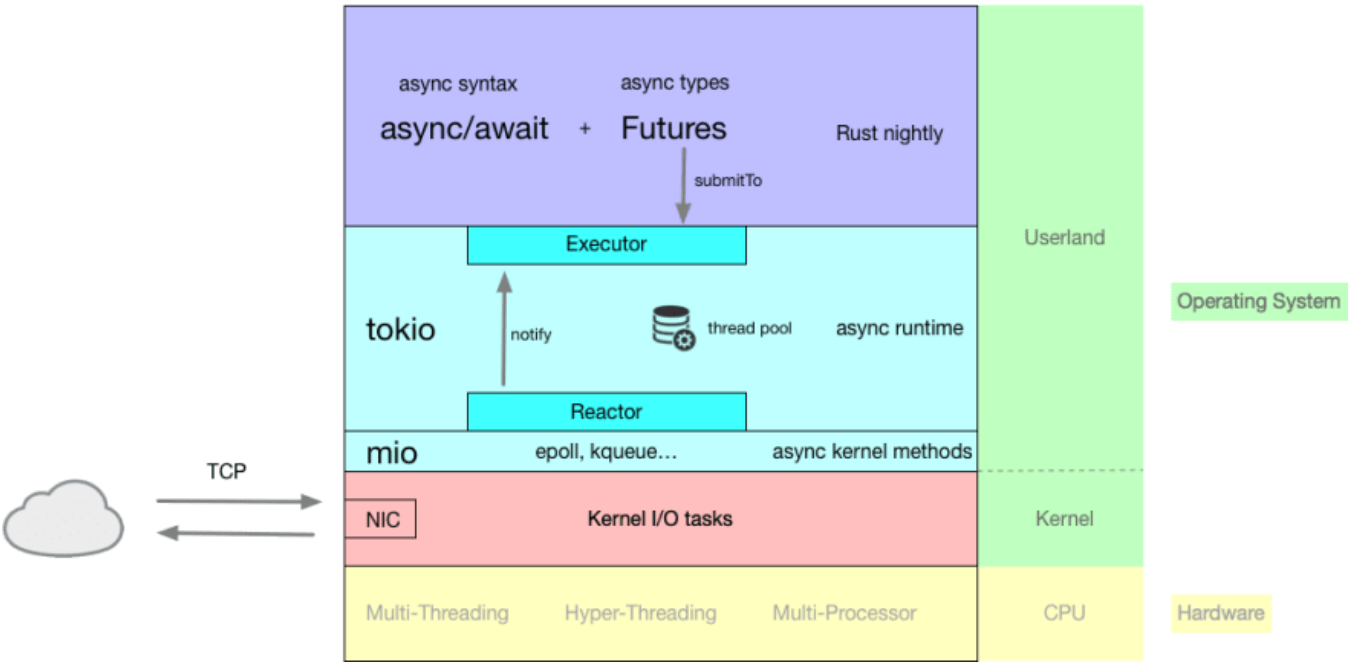

Figura 4: Esquema de funcionamiento del runtime de tokio

Para utilizar Tokio debemos incluirlo en las dependencias del fichero de configuración cargo.toml:

```
tokio = { version = "1", features = ["full"] }
```

El parámetro "full" significa que estamos incluyendo todos los paquetes del runtime.

4.2.1. Tareas

El módulo **tokio::task** proporciona herramientas importantes para trabajar con tareas:

- La función **spawn** y el tipo **JoinHandle**, se utiliza para programar un nueva tarea en el runtime de Tokio y esperar el resultado de una tarea generada, respectivamente.

El módulo **tokio::sync** contiene las primitivas de sincronización para usarlo cuando necesitamos comunicar o compartir datos, incluyendo:

- **Channels** (oneshot, mpsc, and watch) para el envío de variables entre tareas.
- **Mutex**, para controlar el acceso compartido y valores mutables.
- Un tipo de **Barrera asíncrona**, para que varias tareas se sincronicen antes de iniciar un cálculo.

4.2.2. Ejemplos

```rust
use tokio::io::AsyncWriteExt;
use tokio::net::TcpStream;
use std::error::Error;

#[tokio::main] // incluir esta macro para poder ejecutar nuestro código en el runtime de Tokio
pub async fn main() -> Result<(), Box<dyn Error>> {
    // Tokio TcpStream, completamente asíncrono.
    let mut stream = TcpStream::connect("127.0.0.1:6142").await?;
    println!("created stream");

    let result = stream.write(b"Hola mundo\n").await;
    println!("wrote to stream; success={:?}", result.is_ok());

    Ok(()) // enum Result, contiene el valor si todo ha ido bien
}
```

En este ejemplo se muestra un cliente simple que abre una conexión TCP usando el módulo de Tokio **tokio::net::TcpStream**, escribe "Hola mundo" y cierra la conexión.

```rust
use tokio::{
    io::{AsyncBufReadExt, AsyncWriteExt, BufReader},
    net::TcpListener,
    sync::broadcast,
};

#[tokio::main]
async fn main() {
    let listener = TcpListener::bind("localhost:8080").await.unwrap(); // objeto de E/S
    // que representa un socket TCP y escucha las conexiones entrantes

    let (tx, _rx) = broadcast::channel(10); // comunicación entre los clientes utilizando channels
    // para el envío de mensajes entre las tareas. tx = emisor y rx = receptor

    loop {
        let (mut socket, addr) = listener.accept().await.unwrap(); // aceptar nuevas conexiones entrantes
        // devuelve una tupla Result<(TcpStream, SocketAddr)>

        let tx = tx.clone(); // clonamos la variable tx para poder usarla dentro del bloque asíncrono
        let mut rx = tx.subscribe(); // crea un nuevo receptor

        tokio::spawn(async move { // macro spawn para generar tareas
            let (read, mut writer) = socket.split(); // divide un TcpStream en: mitad de lectura y mitad de escritura
            let mut reader = BufReader::new(read); // Buffer de lectura, le pasamos la mitad de lectura de tipo stream
            let mut line = String::new();

            loop {
                tokio::select! { // macro select para ejectuar los futures de lectura y escritura simultáneamente
                // identificador = future => codigo
                    result = reader.read_line(&mut line) => {
                        if result.unwrap() == 0 {
                            break; // cuando un cliente se desconecta, salimos del bucle
                        }

                        tx.send((line.clone(), addr)).unwrap(); //envia los datos a los receptores activos
                        line.clear();
                    }
                    result = rx.recv() => {
                        let (msg, other_addr) = result.unwrap();
                        if addr != other_addr {
                            writer.write_all(&msg.as_bytes()).await.unwrap(); //escribir el mensaje en cada receptor
                        }
                    }
                }
            }
        });
    }
}
```

Este ejemplo es algo más completo que el anterior, se trata de un chat server en el que aparecen módulos mencionados anteriormente como:

- La macro **tokio::spawn** para tener múltiples clientes.
- El módulo **tokio::sync::broadcast** para la comunicación entre emisor y receptor.
- La macro **tokio::select!** para ejecutar múltiples tareas asíncronas al mismo tiempo

5. Conclusiones

Una vez finalizado la redacción del documento, he podido observar que la programación asíncrona se utiliza especialmente para aplicaciones en internet del estilo cliente / servidor para tratar la posibilidad de múltiples peticiones simultáneamente.

Este forma de programación en Rust me ha resultado, sobretodo al principio, un poco confuso de entender por el simple hecho de su funcionamiento, pero finalmente se ha entendido. Me he encontrado con la diferencia de await con respecto al resto de lenguajes como JavaScript y es que en Rust cuando ejecutamos una función asíncrona ésta no está asociada con ningún bucle de eventos (Event Loop) como JS si no que espera a que alguien la ejecute, un future no hace nada hasta que no se activa mediante .await.

Otra peculiaridad del lenguaje es que para poder ejecutar una aplicación asíncrona necesitamos de bibliotecas externas conocidas como runtimes, como el mencionado en la sección anterior.

Referencias

[1] Rust-lang Github.io. *Getting started - asynchronous programming in rust.* 2021. URL: `https://rust-lang.github.io/async-book/`.

[2] J Narvaez. *Concurrencia. Gitbooks.io.* URL: `https://goyox86.gitbooks.io/el-libro-de-rust/content/concurrency.html`.

[3] Wikipedia contributors. *Rust (lenguaje de programación).* 2021. URL: `https://es.wikipedia.org/wiki/Rust_(lenguaje_de_programaci%C3%B3n)`.

[4] K Wróbel. *Rust programming language - what is rust used for and why is so popular?* 2021. URL: `https://codilime.com/blog/why-is-rust-programming-language-so-popular/`.

[5] Rust-lang.org. *Future in std::future - Rust.* 2021. URL: `https://doc.rust-lang.org/std/future/trait.Future.html`.

[6] Gitbook.io. *The reactor-executor pattern.* 2021. URL: `https://cfsamsonbooks.gitbook.io/epoll-kqueue-iocp-explained/appendix-1/reactor-executor-pattern`.

[7] Docs.rs. *Tokio - Rust.* 2021. URL: `https://docs.rs/tokio/latest/tokio/`.

[8] Docs.rs. *async_std - Rust.* 2021. URL: `https://docs.rs/async-std/latest/async_std/`.