Async Rust: compartiendo memoria

Metodos para manejar estado mutable compartido en Async Rust

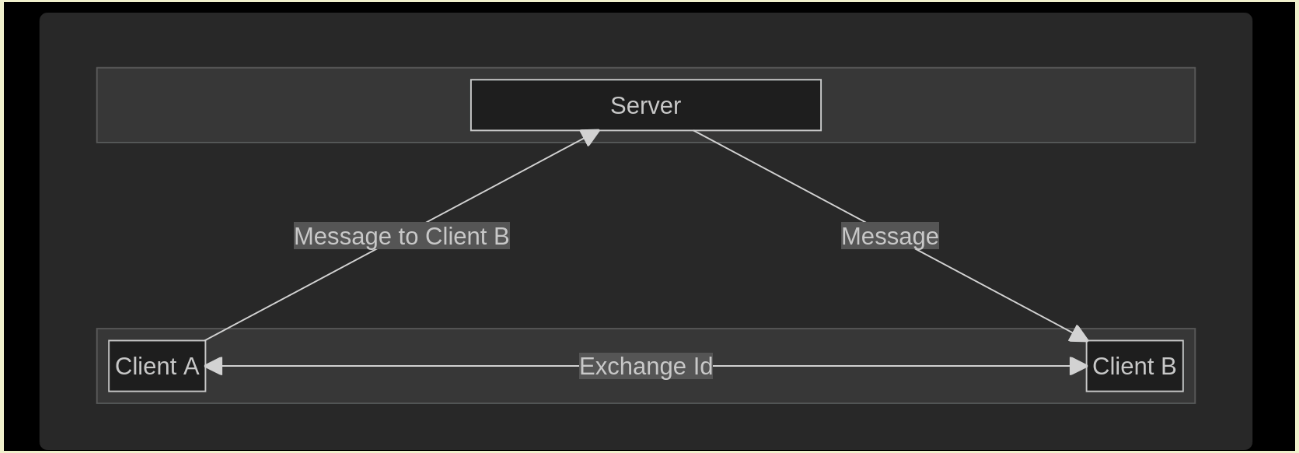
27/11/2025

Gabriel A. Steinberg

conectado taping-memory.dev dusklabs.xyz @tapingmemory.bsky.social

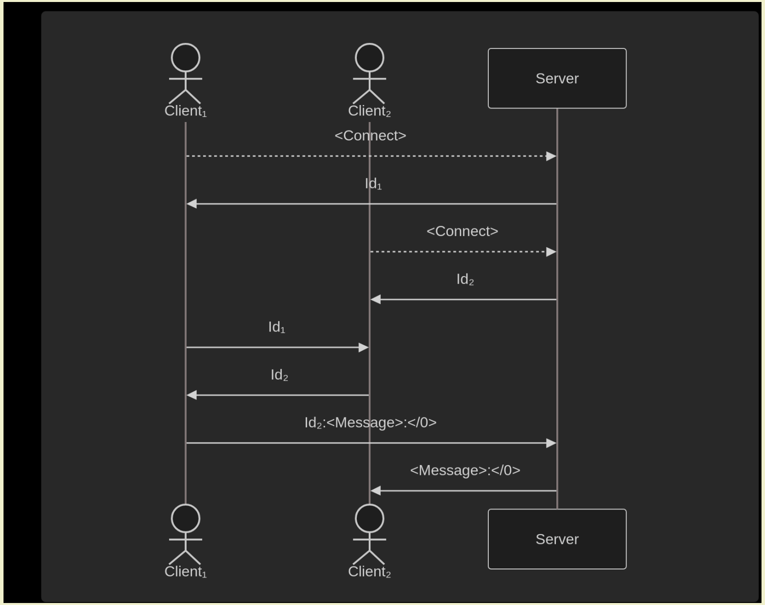
1 Un server TCP

Protocolo



- El servidor le asigna un ID a cada cliente
- Los clientes se informan su ID por un canal fuera de banda
- Cada cliente le puede pedir al servidor que envie un mensaje

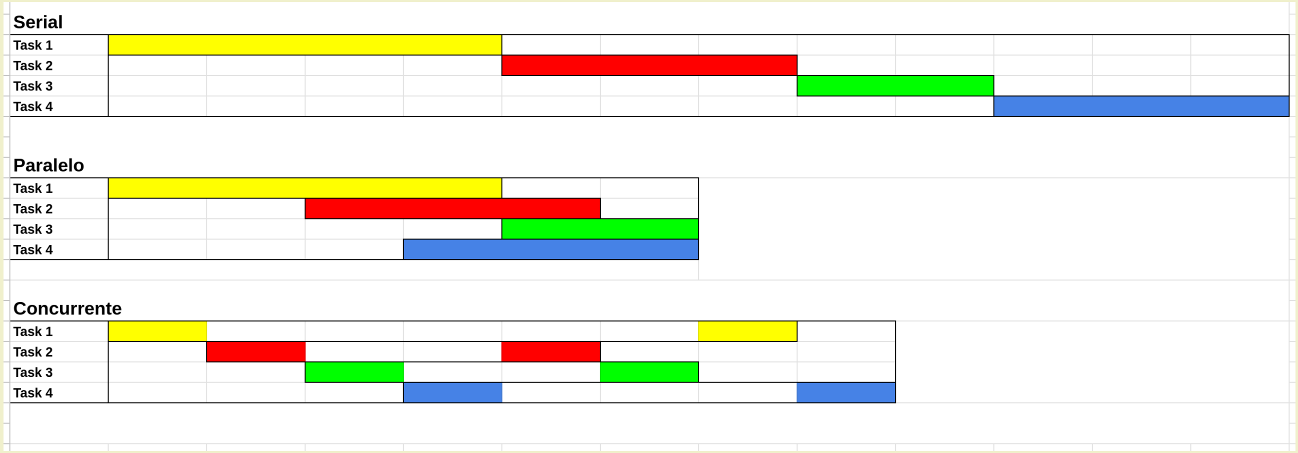
Secuencia



Client$_1$
Client$_2$
Server
<Connect>
Id$_1$
<Connect>
Id$_2$
Id$_1$
Id$_2$
Id$_2$:<Message>:</0>
<Message>:</0>
Client$_1$
Client$_2$
Server

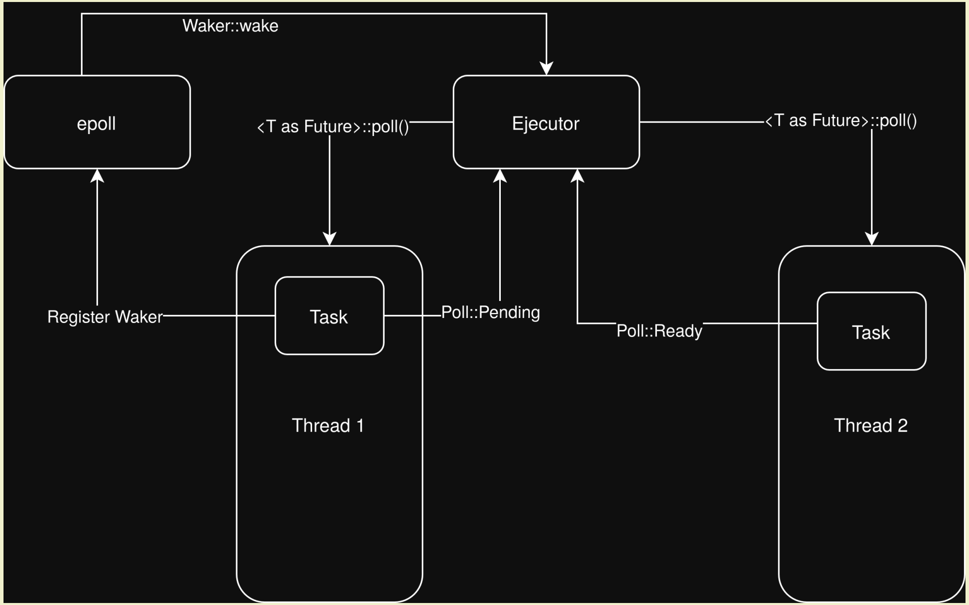
Async

- Esperar eventos concurrentemente
- Ejecutar tareas paralelamente



El server necesita concurrencia

Un modelo mental util



2 Implementacion

Un Server

```rust
struct Server {
    listener: tokio::net::TcpListener,
    connections: HashMap<u32, tokio::net::TcpStream>,
}
```

Los tipos pueden llegar a cambiar pero se mantiene esta idea.

Importante: Estamos usando la version async (de tokio) de `TcpListener` y `TcpStream`

Implementacion Naive (no compila)

```rust
impl Server {
    pub async fn handle_connections(&mut self) {
        loop {
            let (socket, _) = self.listener.accept().await.unwrap();

            tokio::spawn(async move {
                self.handle_connection(socket).await;
            });
        }
    }

    async fn handle_connection(&mut self, mut socket: TcpStream) {...}
}
```

El error

```rust
impl Server {
    pub async fn handle_connections(&mut self) {
        loop {
            let (socket, _) = self.listener.accept().await.unwrap();

            tokio::spawn(async move {
                self.handle_connection(socket).await;
            });
        }
    }

    async fn handle_connection(&mut self, mut socket: TcpStream) {...}
}
```

} (1)

borrow escapes function

Spawn

```rust
pub fn spawn<F>(future: F) -> JoinHandle<F::Output> where
    F: Future + Send + 'static,
    F::Output: Send + 'static,
```

- El `Future` a ejecutar tiene que ser `'static` porque va a vivir mas que la funcion donde se llama.
- El borrow mutable a `self` continua viviendo despues de que termina la iteracion del `loop` por lo que rompe las reglas del borrow checker, si queremos mantener concurrencia.

Conclusion: Necesitamos algun mecanismo de sincronizacion.

Implementacion con Mutex

Vamos a usar un `tokio::sync::Mutex` porque vamos a necesitar tener el lock mientras llamamos await.

```rust
async fn handle_connection(&self, mut socket: TcpStream) {
  ...
  let (dest, msg) = parse_message(&mut buffer);
  self.connections.lock().get(dest).send(msg).await;
  ...
}
```

Nuevo server

```rust
struct Server {
    listener: TcpListener,
    connections: tokio::sync::Mutex<HashMap<u32, TcpStream>>,
}
```

```rust
    pub async fn handle_connections(self: Arc<Self>) {
        loop {
            let (socket, _) = self.listener.accept().await.unwrap();
            let server = Arc::clone(&self);

            tokio::spawn(async move { server.handle_connection(socket).await; });
        }
    }
```

Handle Connections (Mutex)

```rust
async fn handle_connection(&self, mut socket: TcpStream) {
    ...
    self.connections.insert(id, socket);
    loop {
         let connection = self.connections.lock().await.get_mut(&id).unwrap();
         connection.read_buf(&mut buffer).await

        let Ok((dest, m)) = parse_message(&mut buffer) else { continue; };

        let mut connections = self.connections.lock().await;

        connections.get_mut(&dest).write_all(&m).await;
    }
}
```

Deadlock

```rust
async fn handle_connection(&self, mut socket: TcpStream) {
    ...
    self.connections.insert(id, socket);
    loop {
         let connection = self.connections.lock().await.get_mut(&id).unwrap();
         connection.read_buf(&mut buffer).await

        let Ok((dest, m)) = parse_message(&mut buffer) else { continue; };

        let mut connections = self.connections.lock().await;

        connections.get_mut(&dest).write_all(&m).await;
    }
}
```

La solucion

```rust
struct Server {
    listener: TcpListener,
    connections: tokio::sync::Mutex<HashMap<u32, OwnedWriteHalf>>,
}
```

Handle Connection

```rust
async fn handle_connection(&self, mut socket: TcpStream) {
    ...
    let (mut read, mut write) = socket.into_split();
    self.connections.insert(id, write);
    loop {
         let connection = read.await.get_mut(&id).unwrap();
         connection.read_buf(&mut buffer).await

        let Ok((dest, m)) = parse_message(&mut buffer) else { continue; };

        let mut connections = self.connections.lock().await;
        connections.get_mut(&dest).write_all(&m).await;
    }
}
```

Aun asi...

```rust
async fn handle_connection(&self, mut socket: TcpStream) {
    ...
    let (mut read, mut write) = socket.into_split();
    self.connections.insert(id, write);
    loop {
         let connection = read.await.get_mut(&id).unwrap();
         connection.read_buf(&mut buffer).await

        let Ok((dest, m)) = parse_message(&mut buffer) else { continue; };

        let mut connections = self.connections.lock().await;
        connections.get_mut(&dest).write_all(&m).await;
    }
}
```

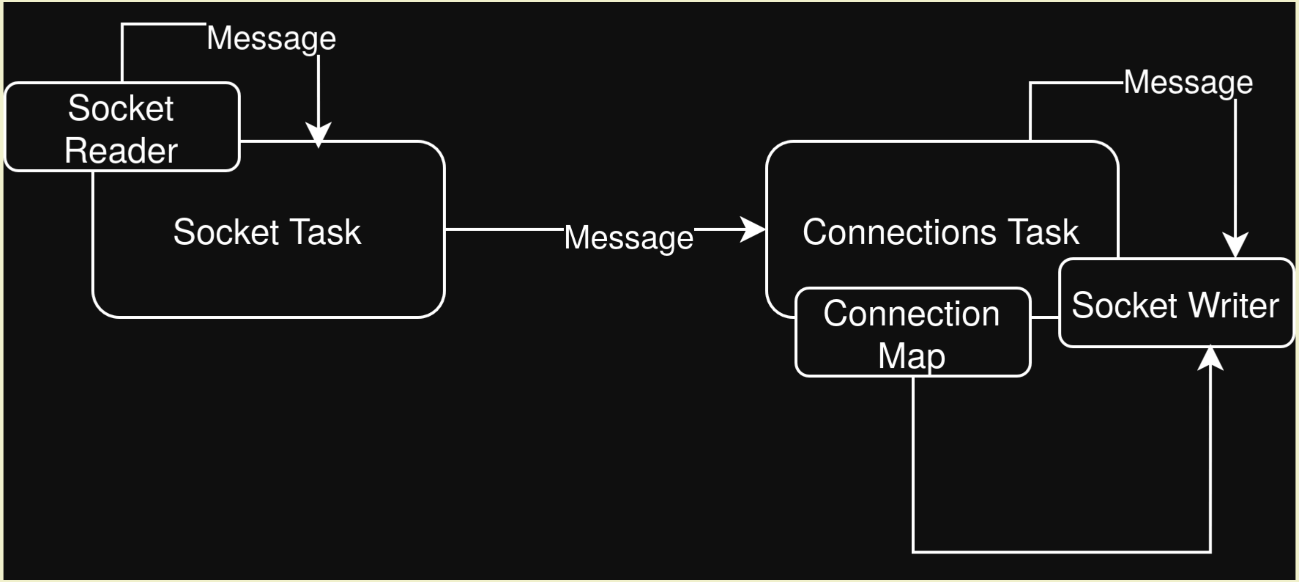
Los mutex son dificiles :(

- Un buffer lleno previene a cualquier otro cliente avanzar
- Podemos usar `Arc<Mutex<OwnedWriteHalf>>` pero es mas complejo aun!

Veamos otra manera

3 Canales

Idea general



Implementacion

```rust
async fn handle_connection(&self, mut socket: TcpStream) {
    ...
    let (mut read, mut write) = socket.into_split();
    self.tx.send(Message::new(id, write)).await;

    loop {
         let connection = read.await.get_mut(&id).unwrap();
         connection.read_buf(&mut buffer).await

        let Ok((dest, m)) = parse_message(&mut buffer) else { continue; };
        self.tx.send(Message::Send(dest, m)).await;
    }
}
```

Message dispatcher

```rust
async fn message_dispatcher(mut rx: mpsc::Receiver<Message>) {
    let mut connections = HashMap::new();
    while let Some(msg) = rx.recv().await {
        match msg {
            Message::New(id, mut connection) => {
                connection.write_u32(id).await;
                connections.insert(id, connection);
            }
            Message::Send(id, items) => {
                connections.get_mut(&id).write_all(&items).await;
            }
        }
    }
}
```

Aun asi puede bloquear...

```rust
async fn message_dispatcher(mut rx: Receiver<Message>) {
    let mut connections = HashMap::new();
    while let Some(msg) = rx.recv().await {
        match msg {
            Message::New(id, mut connection) => {
                connection.write_u32(id).await;
                connections.insert(id, connection);
            }
            Message::Send(id, items) => {
                connections.get_mut(&id).write_all(&items).await;
            }
        }
    }
}
```

Una mejora

```rust
async fn message_dispatcher(mut rx: Receiver<Message>) {
    let mut connections = HashMap::new();
    while let Some(msg) = rx.recv().await {
        match msg {
            Message::New(id, mut connection) => {
                let (tx, rx) = mpsc::channel(100);
                tokio::spawn(client_dispatcher(connection, rx))
                ...
            }
            Message::Send(id, items) => {
                connections.get_mut(&id).send(items).await;
            }
        }
    }}
```

Una mejora (ii)

```rust
async fn client_dispatcher(mut connection: OwnedWriteHalf, mut rx: Receiver<Bytes>) {
    loop {
        let m = rx.recv().await.unwrap();
        if let Err(e) = connection.write_all(&m).await {
            eprintln!("Failed to write to socket: {e}");
            break;
        }
    }
}
```

Backpressure

```rust
async fn central_dispatcher(mut rx: Receiver<Message>) {
        ...
            Message::Send(id, items, permission_token) => {
                ...
                connection.send(items).await;
                let _ = permission_token.send(());
            }
        ...
}
```

Backpressure (ii)

```rust
async fn handle_connection(&self, socket: TcpStream) {
    ...

    loop {
        read.read_buf(&mut buffer).await

        ...

        let (permission_token, permission_listener) = oneshot::channel();
        self.tx.send(Message::Send(dest, m, permission_token)).await.unwrap();
        permission_token.recv().await;
    }
}
```

Y errores?

```rust
async fn central_dispatcher(error_tx: Sender<Message>, mut rx: Receiver<Message>) {
    ...
    while let Some(msg) = rx.recv().await {
        match msg {
            Message::New(id, connection) => {
                let (tx, rx) = mpsc::channel(100);
                tokio::spawn(client_dispatcher(connection, rx, error_tx.clone()));
                ...
            }
            Message::Error(e, src) => {
                ...
                connection.get(&src).send(error).await;
            }
        }}}
```

Y errores? (ii)

```rust
async fn client_dispatcher(
    mut connection: OwnedWriteHalf,
    mut rx: mpsc::Receiver<(Bytes, Option<u32>)>,
    errors_tx: mpsc::Sender<Message>,
) {
    loop {
        let (m, src) = rx.recv().await.unwrap();
        if let Err(e) = connection.write_all(&m).await {
            let _ = errors_tx.send(Message::Error(e, src)).await;
            break;
        }
    }
}
```

El problema?

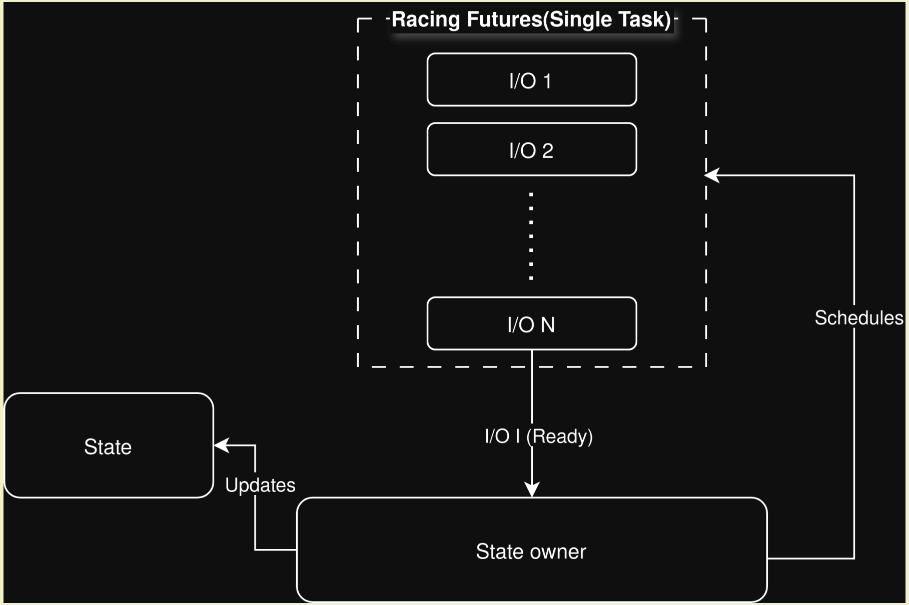
- La complejidad crece, con cada interaccion entre `Task`.
- Manejar errores no es natural necesitas agregar nuevos mecanismos de comunicacion.
- Es dificil de debuggear.

Dado que el stack trace no sigue la forma natural de function calls, rastrear de donde proviene un valor se vuelve muy complicado.

Pero podemos simplificar esto, sacrificando el paralelismo.

4 Implementacion en un solo task

Idea general



Nueva interfaz para sockets

```rust
struct ReadSocket {
    buffer: BytesMut,
    reader: OwnedReadHalf,
    id: u32,
}
```

Nueva interfaz para sockets (ii)

```rust
impl ReadSocket {
    async fn read(&mut self) -> Result<Event> {
        loop {
            if let Ok(read_result) = parse_message(&mut self.buffer) {
                return Ok(Event::Read(read_result));
            };

            self.reader.read_buf(&mut self.buffer)
                .await
                .map_err(|e| (e, self.id))?;
        }
    }
}
```

Nueva interfaz para sockets (iii)

```rust
struct WriteSocket {
    buffers: VecDeque<Bytes>,
    writer: OwnedWriteHalf,
    id: u32,
}
```

```rust
impl WriteSocket {
    fn send(&mut self, buf: Bytes) {
        self.buffers.push_back(buf);
    }
}
```

Nueva interfaz para sockets (iv)

```rust
impl WriteSocket {
    async fn advance_send(&mut self) -> Result<Event> {
        loop {
            let Some(buffer) = self.buffers.front_mut() else {
                std::future::pending::<Infallible>().await;
                unreachable!();
            };

            self.writer.write_all_buf(buffer)
                .await
                .map_err(|e| (e, self.id))?;

            self.buffers.pop_front();
}}}
```

Server

```rust
struct Server {
    read_connections: HashMap<u32, ReadSocket>,
    write_connections: HashMap<u32, WriteSocket>,
    listener: TcpListener,
}
```

Carrera de futuros

```rust
fn next_event(&mut self) -> impl Future<Output = Result<Event>> {
    let listen = self.listener.accept()
        .map(|stream| Ok(Event::Connection(stream.unwrap().0)));

    let reads = self.read_connections.values_mut()
        .map(|reader| reader.read()).collect::<Vec<_>>()
        .race();

    let writes = self.write_connections.values_mut()
        .map(|write| write.advance_send()).collect::<Vec<_>>()
        .race();

    (listen, reads, writes).race().boxed()
}
```

Loop principal

```rust
pub async fn handle_connections(&mut self) {
    loop {
        ...
        match self.next_event().await {
            Event::Read((dest, items)) => {
                self.write_connections.get_mut(&dest).send(items);
            }
            Event::Connection(tcp_stream) => {
                ...
                self.read_connections.insert(id, ReadSocket::new(r, id));
                self.write_connections.insert(id, WriteSocket::new(w, id));
            }
        }
    }}
```

Backpressure

```rust
fn next_event(&mut self) -> impl Future<Output = Result<Event>> {
    ...

    if self.buffer_size() > MAX_BUFFER {
        (listen, writes).race().boxed()
    } else {
        (listen, reads, writes).race().boxed()
    }
}
```

Errores

```rust
pub async fn handle_connections(&mut self) {
    loop {
        let ev = match self.next_event().await {
          Ok(ev) => ev,
          Err((e, sender)) => {
              self.write_connections.get(&sender).send(error);
              continue;
          }
        };
        ...
    }
}
```

En resumen

- Flujo natural de errores
- Acceso `&mut` al estado
- Facil de trazar el origen de los valores
- Facil de razonar

Pero podriamos tener mas?

- Necesitamos escribir los adaptadores para los combinadores
- Tenemos que multiplexar/demultiplexar los errores
- El stack trace del IO puede ser dificil de entender
- Los futuros del IO no pueden compartir acceso `&mut` a la misma memoria

5 Conclusiones

Que vimos?

- Trade-offs: Mutex vs Canales vs Race/Select
- Evaluar si necesitamos paralelismo

Consejos

- Empezar por concurrencia sin paralelismo
- En caso de necesitar paralelismo usar canales
- Mutex se necesita en casos muy especificos
- Multiples threads no siempre mejora la performance

Mas alla de esta charla

- Mi post
- Let future be futures
- Tree structured concurrency part I
- Tree structure concurrency part II
- sans-IO: The secret to effective Rust for network services

6 Dudas? Comentarios? Quejas?