Con `std::thread`: El código bloquea cada hilo mientras espera y maneja la conexión. Aunque puede manejar múltiples conexiones concurrentemente, cada nueva conexión requiere un hilo nuevo. En sistemas con muchos hilos, puede resultar ineficiente.

```
use std::{net, thread};


let listener = net::TcpListener::bind(address)?;
for socket_result in listener.incoming() {
    let socket = socket_result?;
    let groups = chat_group_table.clone();
    thread::spawn(|| {
        log_error(serve(socket, groups));
    });
}
```

Con `async-std`: Utiliza un solo hilo para manejar muchas conexiones concurrentemente, gracias a la programación asíncrona. Esto permite que el servidor escale mejor bajo una carga pesada de conexiones, ya que no está limitado por la creación de hilos para cada conexión.

```
use async_std::{net, task};

let listener = net::TcpListener::bind(address).await?;
let mut new_connections = listener.incoming();
while let Some(socket_result) = new_connections.next().await {
    let socket = socket_result?;
    let groups = chat_group_table.clone();
    task::spawn(async {
        log_error(serve(socket, groups).await);
    });
}
```

El trait Future gestiona el estado de las operaciones asincrónicas.

```
trait Future {
    type Output;

    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) ->
Poll<Self::Output>;
}
```

- El tipo `Output` representa el resultado de una tarea asincrónica.
- La función `poll` comprueba el estado de la operación.

```
enum Poll<T> {
    Ready(T),
    Pending,
}
```

- Ready: terminó la tarea.
- Pending: Aún no terminó.

Versión sincrónica

```
fn read_to_string(&mut self, buf: &mut String) ->
std::io::Result<usize>;
```

La función bloquea la ejecución del hilo actual mientras lee el contenido. Esto significa que **el hilo que invoca esta función no puede hacer nada más hasta que se haya completado la lectura del recurso.** Si el recurso tarda en responder (por ejemplo, al leer un archivo grande o esperar una respuesta de una red), el hilo estará bloqueado durante ese tiempo.
Devuelve un `Result` que contiene el número de bytes leídos o un error en caso de que la operación falle.

Versión asincrónica

```
fn read_to_string(&mut self, buf: &mut String) -> impl Future<Output =
Result<usize>>;
```

La función no bloquea el hilo que la invoca. El hilo puede seguir ejecutando otras tareas mientras espera que la operación de lectura se complete. Este enfoque es útil cuando necesitas realizar múltiples operaciones I/O concurrentemente, ya que permite que otras tareas se ejecuten mientras se esperan resultados de operaciones I/O.
Devuelve un **futuro** (`impl Future`). Este futuro representará una operación asincrónica que se completará en algún momento en el futuro.

```rust
async fn cheapo_request(host: &str, port: u16, path: &str) ->
std::io::Result<String> {
    // Cuerpo de la función
}
```

Función async -> devuelve Future

```rust
let mut socket = net::TcpStream::connect((host, port)).await?;
```

Await -> permite esperar a que un futuro se complete sin bloquear el hilo (solo se usa en funciones async).

Flujo de ejecución en funciones async y await

1. **Invocación de una función async:**

- Cuando llamas a una función async, no se ejecuta inmediatamente. En cambio, devuelve un Future.

2. **Primer poll:**

- Cuando un futuro se pollea por primera vez, comienza la ejecución del cuerpo de la función async hasta que se encuentra una expresión await.
- En el caso de la función cheapo_request, la ejecución se detendría en el primer await (cuando intentamos conectar con el servidor).

3. **El futuro se suspende (si es necesario):**

- Si la operación asincrónica no está lista, como la conexión a un servidor, el poll devuelve Poll::Pending.
- El sistema de ejecución asincrónica guarda el estado local de la función (el punto donde se detuvo la ejecución), lo que permite que se reanude después.

4. **await:**

- await toma ownership del futuro y hace el polling de este futuro.

- Si el futuro está listo (es decir, si la operación de conexión a la red está completa), el futuro retorna Poll::Ready(value) y continúa la ejecución del cuerpo de la función async.
- Si el futuro aún no está listo, el await devuelve Poll::Pending, y la función se suspende hasta que se pueda reanudar.

5. **Reanudación después de await:**

- En la próxima invocación de poll, el futuro sabe exactamente desde qué punto debe continuar. Esto se debe a que el futuro almacena el estado local de la función, es decir, las variables y el punto exacto donde se detuvo la ejecución.
- Así que la invocación a poll después del await continuará desde el punto donde se hizo el await, por ejemplo, después de que se establezca la conexión TCP.

```
use async_std::io::prelude::*;
use async_std::net;

async fn cheapo_request(host: &str, port: u16, path: &str) ->
std::io::Result<String> {
    // Se conecta de manera asincrónica al servidor
    let mut socket = net::TcpStream::connect((host, port)).await?;

    // Se crea una solicitud HTTP
    let request = format!("GET {} HTTP/1.1\r\nHost: {}\r\n\r\n", path,
host);

    // Escribe la solicitud al servidor de manera asincrónica
    socket.write_all(request.as_bytes()).await?;

    // Finaliza la escritura de la solicitud
    socket.shutdown(net::Shutdown::Write)?;

    // Lee la respuesta del servidor de manera asincrónica
    let mut response = String::new();
    socket.read_to_string(&mut response).await?;

    Ok(response)
}
```

`block_on` -> Bloquea un hilo hasta que un futuro se haya completado y obtenga su resultado.

```
fn block_on<F: Future>(future: F) -> F::Output

use async_std::task;

fn main() -> std::io::Result<()> {
    let response = task::block_on(cheapo_request("example.com", 80,
"/"))?;
    println!("{}", response);
    Ok(())
}
```