

SNU SCSC + WaffleStudio 스터디

Week 12: Asynchronous Programming, Part 3

Chris Ohk

utilForever@gmail.com

- 비동기식 클라이언트와 서버
 - 서버의 메인 함수
 - 채팅 연결 처리하기 : 비동기 뮤텍스
 - 그룹 테이블 : 동기 뮤텍스
 - 채팅 그룹 : tokio의 브로드캐스트 채널
- 기본 제공 뮤타와 이그제큐터

서버의 메인 함수

SNU SCSC + WaffleStudio 스터디
Week 12: Asynchronous Programming, Part 3

- 채팅 서버의 메인 파일인 `main.rs`를 보자.

```
● ● ●

use async_chat::utils::ChatResult;
use async_std::prelude::*;

use std::sync::Arc;

mod connection;
mod group;
mod group_table;

use connection::serve;
```

서버의 메인 함수

SNU SCSC + WaffleStudio 스터디
Week 12: Asynchronous Programming, Part 3

- 채팅 서버의 메인 파일인 `main.rs`를 보자.

```
fn main() -> ChatResult<()> {
    let address = std::env::args().nth(1).expect("Usage: server ADDRESS");
    let chat_group_table = Arc::new(group_table::GroupTable::new());

    async_std::task::block_on(async {
        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);
            });
        }
    })
}
```

서버의 메인 함수

SNU SCSC + WaffleStudio 스터디
Week 12: Asynchronous Programming, Part 3

- 채팅 서버의 메인 파일인 `main.rs`를 보자.



```
● ● ●

fn log_error(result: ChatResult<()>) {
    if let Err(error) = result {
        eprintln!("Error: {error}");
    }
}
```

- **코드 설명**

- 약간의 설정 작업을 하고 나서 `block_on`을 호출해 실제 작업을 하는 `async` 블록을 생성한다.
- 먼저 클라이언트에서 오는 연결을 처리하기 위해서 `TcpListener` 소켓을 만든다.
이 소켓의 `incoming` 메소드는 `std::io::Result<TcpStream>` 값의 스트림을 반환한다.
- 이어서 들어오는 연결마다 `connection::serve` 함수를 실행하는 비동기 태스크를 띄운다.
각 태스크는 `GroupTable` 값의 레퍼런스를 받는데, 이는 서버의 현재 채팅 그룹 목록을 표현하는 값으로
레퍼런스 카운트 기반의 포인터인 `Arc`를 통해서 모든 연결이 공유한다.
- `connection::serve` 함수가 오류를 반환하면 표준 오류 출력에 메시지를 기록하고 태스크를 종료한다.
다른 연결은 계속 그대로 실행된다.

채팅 연결 처리하기

SNU SCSC + WaffleStudio 스터디
Week 12: Asynchronous Programming, Part 3

- 다음은 서버의 핵심 요소인 connection 모듈의 serve 함수다.

```
● ● ●

use async_chat::utils::{self, ChatResult};
use async_chat::{FromClient, FromServer};
use async_std::io::BufReader;
use async_std::net::TcpStream;
use async_std::prelude::*;
use async_std::sync::Arc;
use async_std::sync::Mutex;

use crate::group_table::GroupTable;
```

채팅 연결 처리하기

SNU SCSC + WaffleStudio 스터디
Week 12: Asynchronous Programming, Part 3

- 다음은 서버의 핵심 요소인 connection 모듈의 serve 함수다.



```
pub async fn serve(socket: TcpStream, groups: Arc<GroupTable>) -> ChatResult<()> {
    let outbound = Arc::new(Outbound::new(socket.clone()));
    let buffered = BufReader::new(socket);
    let mut from_client = utils::receive_as_json(buffered);

    while let Some(request_result) = from_client.next().await {
        let request = request_result?;
        let result = match request {
            FromClient::Join { group_name } => {
                let group = groups.get_or_create(group_name);
                group.join(outbound.clone());
                Ok(())
            }
        }
    }
}
```

채팅 연결 처리하기

SNU SCSC + WaffleStudio 스터디
Week 12: Asynchronous Programming, Part 3

- 다음은 서버의 핵심 요소인 connection 모듈의 serve 함수다.



```
● ● ●

FromClient::Post {
    group_name,
    message,
} => match groups.get(&group_name) {
    Some(group) => {
        group.post(message);
        Ok(())
    }
    None => Err(format!("Group '{group_name}' does not exist")),
},
};

if let Err(message) = result {
    let report = FromServer::Error(message);
    outbound.send(report).await?;
}
}

Ok(())
}
```

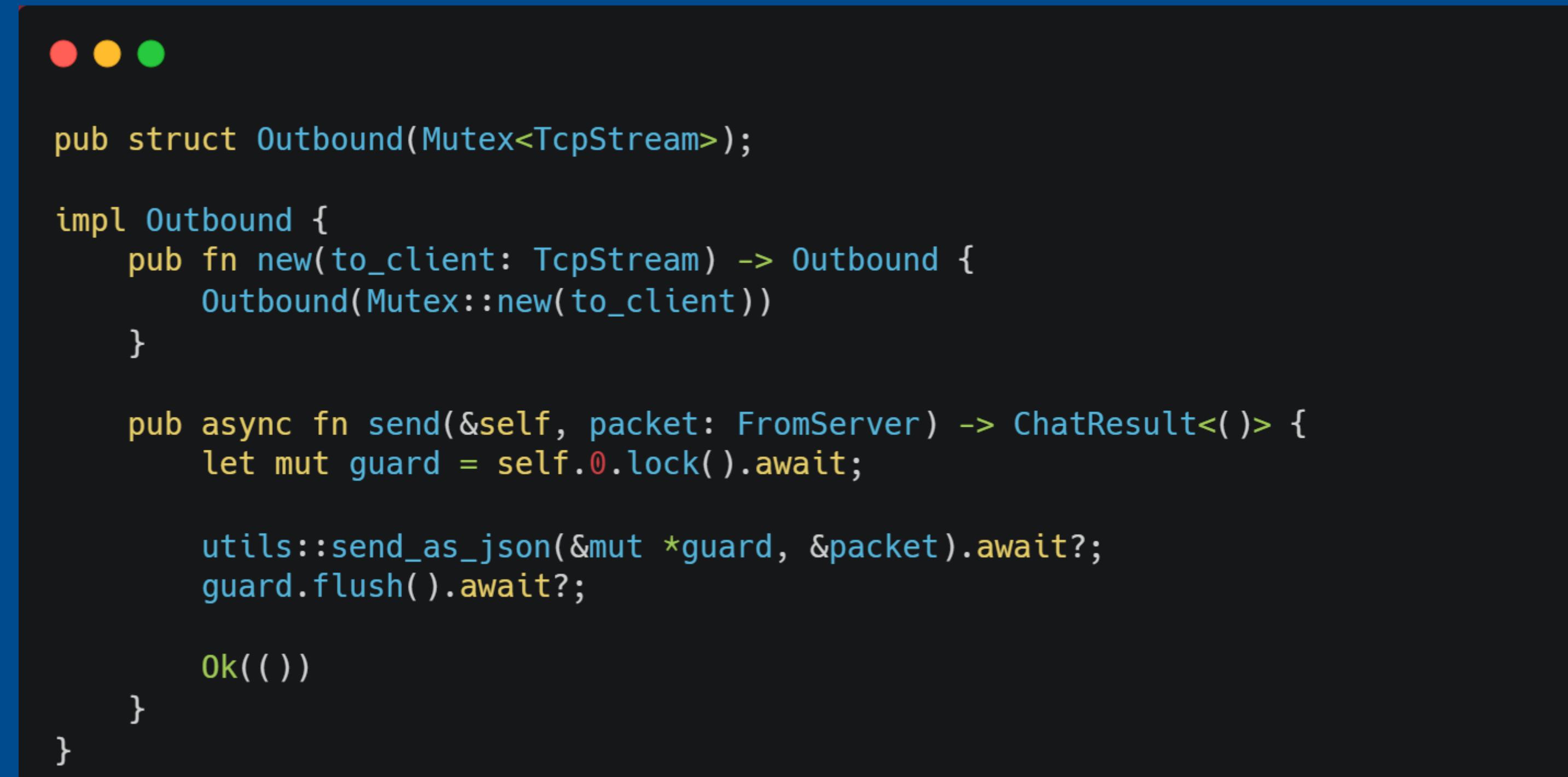
- **코드 설명**

- 코드의 대부분을 차지하는 건 들어오는 `FromClient` 값의 스트림을 처리하는 반복문이다.
- 이 스트림은 버퍼링되는 TCP 스트림을 `receive_as_json`으로 처리해서 만든다.
오류가 발생하면 `FromServer::Error` 패킷을 생성해서 처리를 실패했다고 클라이언트에게 전한다.
- 클라이언트는 오류 메시지와 더불어 자신이 가입한 채팅 그룹에서 오는 메시지도 받아야 하므로
클라이언트의 연결을 각 그룹과 공유해야 한다.
- 이럴 때는 그냥 모두에게 `TcpStream`의 복제본을 건네주면 되는데,
이렇게 하면 두 소스가 동시에 패킷을 소켓에 기록했을 때 출력이 서로 뒤섞여서
클라이언트가 잘못된 JSON을 받게 될 가능성이 있다.
따라서 연결을 동시에 접근해도 안전할 수 있도록 정리가 필요하다.

채팅 연결 처리하기

SNU SCSC + WaffleStudio 스터디
Week 12: Asynchronous Programming, Part 3

- 이 부분은 Outbound 탑입이 관리한다.



The screenshot shows a terminal window with three colored window control buttons (red, yellow, green) at the top. The terminal displays the following Rust code:

```
pub struct Outbound(Mutex<TcpStream>);

impl Outbound {
    pub fn new(to_client: TcpStream) -> Outbound {
        Outbound(Mutex::new(to_client))
    }

    pub async fn send(&self, packet: FromServer) -> ChatResult<()> {
        let mut guard = self.0.lock().await;

        utils::send_as_json(&mut *guard, &packet).await?;
        guard.flush().await?;

        Ok(())
    }
}
```

- 코드 설명
 - Outbound는 값이 생성될 때 TcpStream의 소유권을 가져다가 한 번에 한 태스크만 쓸 수 있도록 Mutex로 감싸 둔다.
 - serve 함수는 클라이언트가 가입한 모든 그룹이 공유된 같은 Outbound 인스턴스를 가리킬 수 있도록 각 Outbound를 레퍼런스 카운트 기반의 포인터인 Arc로 감싸 둔다.
 - Outbound::send 호출은 먼저 뮤텍스를 잠궈 내부에 있는 TcpStream을 역참조하는 가드(Guard) 값을 가져온다. 그런 다음 send_as_json을 써서 packet을 전송하고, 끝으로 guard.flush()를 호출해서 어딘가에 있는 버퍼에 전송되다 만 데이터가 남아 있는 일이 없도록 정리한다.
(사실 TcpStream은 데이터를 버퍼링하지 않지만, Write 트레잇이 그런 구현을 허용하므로 만약을 위해 처리해 둔다.)
 - 표현식 &mut *guard를 쓰면 Rust가 트레잇 바운드를 만족하는지 볼 때 Deref 강제 변환을 적용하지 않는다는 사실을 회피할 수 있다. 즉, 뮤텍스 가드를 명시적으로 역참조한 다음 보호하고 있는 TcpStream의 변경할 수 있는 레퍼런스를 빌려 오는 식으로 send_as_json이 요구하는 &mut Stream을 산출하는 것이다.

- Outbound는 `std::sync::Mutex`가 아니라 `async_std::sync::Mutex` 타입을 쓴다.
여기에는 세 가지 이유가 있다.
 - 첫 번째로 `std::sync::Mutex`는 태스크가 뮤텍스 가드를 준 채로 중단되면 오작동할 수도 있다.
 - 해당 태스크를 실행하던 스레드가 같은 `Mutex`를 잠그려는 또 다른 태스크를 집어 들면 문제가 터지는데, `Mutex`의 입장에서 보면 이미 자신을 소유하고 있는 스레드가 또 한 번 자신을 잠그려고 하는 셈이기 때문이다.
 - 표준 `Mutex`는 이런 경우를 처리하게끔 설계되지 않아서 패닉이나 교착 상태에 빠진다.
- Rust는 현재 `std::sync::Mutex` 가드의 수명 안에 `await` 표현식이 들어가 있을 때마다 컴파일 시점에 경고를 내보내도록 만드는 작업이 진행중이다.
- `Outbound::send`는 `send_as_json`과 `guard.flush`의 퓨처를 기다리는 동안 락을 주고 있어야 하므로 반드시 `async_std`의 `Mutex`를 써야 한다.

채팅 연결 처리하기

SNU SCSC + WaffleStudio 스터디
Week 12: Asynchronous Programming, Part 3

- Outbound는 `std::sync::Mutex`가 아니라 `async_std::sync::Mutex` 타입을 쓴다.
여기에는 세 가지 이유가 있다.
- 두 번째로 비동기 `Mutex`의 `lock` 메소드는 가드의 퓨처를 반환하기 때문에,
뮤텍스를 잠그려고 기다리는 태스크는 뮤텍스가 준비될 때까지 자신의 스레드를 다른 태스크에게 양보한다.
(뮤텍스가 이미 사용 가능한 상태라면 `lock` 퓨처는 즉시 준비 상태가 되므로 태스크 자체가 중단되는 일이 없다.)
- 반면 표준 `Mutex`의 `lock` 메소드는 락을 획득할 때까지 기다리는 동안 전체 스레드를 꼼짝 못하게 잡아 둔다.
- 앞에 있는 코드는 네트워크에 패킷을 전송하는 동안 뮤텍스를 줘고 있으므로 시간이 꽈 걸릴 수도 있다.

채팅 연결 처리하기

SNU SCSC + WaffleStudio 스터디
Week 12: Asynchronous Programming, Part 3

- Outbound는 `std::sync::Mutex`가 아니라 `async_std::sync::Mutex` 타입을 쓴다.
여기에는 세 가지 이유가 있다.
 - 세 번째로 표준 Mutex는 락을 건 스레드만 락을 풀 수 있다.
 - 이 규칙을 시행하기 위해 표준 Mutex의 가드 타입은 `Send`를 구현하고 있지 않으며, 따라서 다른 스레드로 전송할 수 없다.
 - 이 말은 이런 가드를 줘고 있는 퓨처 자체도 `Send`를 구현하고 있지 않기 때문에 `spawn`에 넘겨서 스레드 풀을 가지고 실행할 수 없다는 뜻으로, `block_on`이나 `spawn_local`을 써서 실행할 수 밖에 없다.
 - `async_std` Mutex용 가드는 `Send`를 구현하고 있으므로 `spawn`을 통해 실행되는 태스크에서 써도 아무 문제가 없다.

채팅 연결 처리하기

SNU SCSC + WaffleStudio 스터디
Week 12: Asynchronous Programming, Part 3

- '비동기 코드에서는 항상 `async_std::sync::Mutex`를 써라'
이 이야기를 하려는 게 아니다. 그렇게 단순하지 않다.
 - 뮤텍스를 줘고 있는 동안 아무것도 기다릴 필요가 없을 때도 있고 락을 오래 줘고 있지 않을 때도 많기 때문에,
이런 경우에는 표준 라이브러리의 `Mutex`가 훨씬 더 효율적일 수 있다.
 - 다음에 볼 채팅 서버의 `GroupTable` 타입이 바로 이런 경우에 해당한다.

그룹 테이블

SNU SCSC + WaffleStudio 스터디
Week 12: Asynchronous Programming, Part 3

- 다음은 group_table.rs의 전체 내용이다.

```
● ● ●

use crate::group::Group;
use std::collections::HashMap;
use std::sync::{Arc, Mutex};

pub struct GroupTable(Mutex<HashMap<Arc<String>, Arc<Group>>>);

impl GroupTable {
    pub fn new() -> GroupTable {
        GroupTable(Mutex::new(HashMap::new()))
    }

    pub fn get(&self, name: &String) -> Option<Arc<Group>> {
        self.0.lock().unwrap().get(name).cloned()
    }

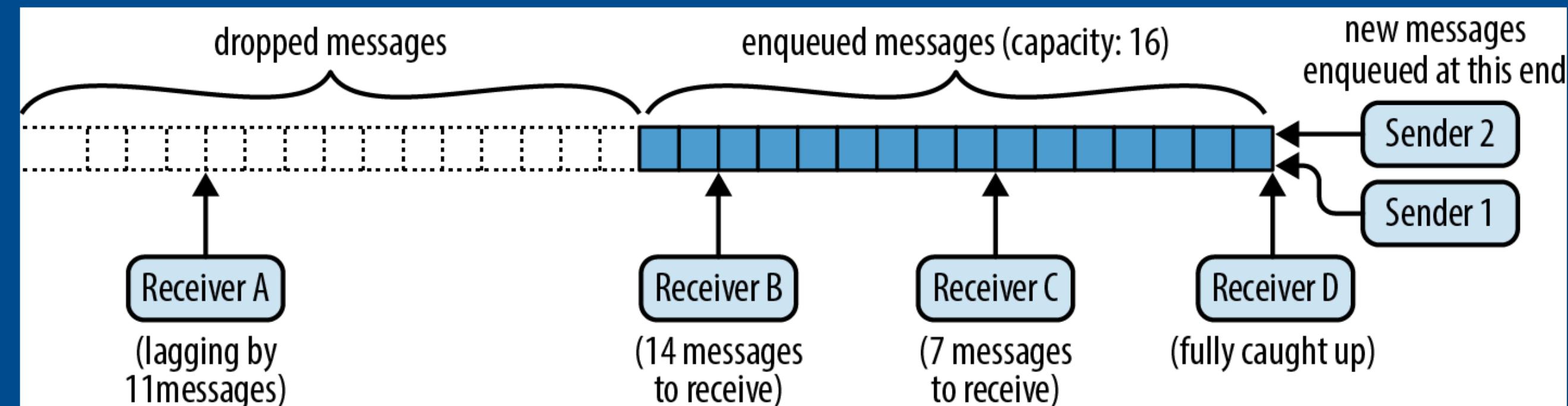
    pub fn get_or_create(&self, name: Arc<String>) -> Arc<Group> {
        self.0
            .lock()
            .unwrap()
            .entry(name.clone())
            .or_insert_with(|| Arc::new(Group::new(name)))
            .clone()
    }
}
```

- **코드 설명**
 - GroupTable은 뮤텍스로 보호되는 간단한 해시 테이블로 채팅 그룹 이름을 실제 그룹에 매핑하며, 둘 다 레퍼런스 카운트 기반의 포인터로 관리한다. get과 get_or_create 메소드는 뮤텍스를 잠그고 몇 가지 해시 테이블 작업을 수행한 뒤 복귀하는데, 어쩌면 이 과정에서 할당이 몇 차례 일어날 수도 있다.
 - GroupTable은 평범하고 오래된 `std::sync::Mutex`를 쓴다.
이 모듈에는 비동기 코드가 전혀 없으므로 피해야 할 `await`도 없다.
만약 여기에 `async_std::sync::Mutex`를 썼다면 `get`과 `get_or_create`를 비동기 함수로 만들어야 했을 테니, 혜택은 적은데 퓨처를 생성하고 중단하고 재개하는 데서 오는 오버헤드만 떠안는 꼴이 됐을 것이다.
뮤텍스는 일부 해시 연산과 어쩌면 있을지도 모를 몇 차례의 할당 과정에서만 잠그면 된다.
 - 채팅 서버의 사용자가 수백만 명으로 불어나서 GroupTable 뮤텍스가 병목이 되면 그땐 이를 비동기로 바꾼다고 해서 문제가 해결되지 않는다. 이럴 때는 `HashMap` 말고 동시 접근에 특화된 다른 타입을 쓰는 게 더 낫다.
예를 들어, `dashmap` 크레이트는 그런 타입을 제공한다.

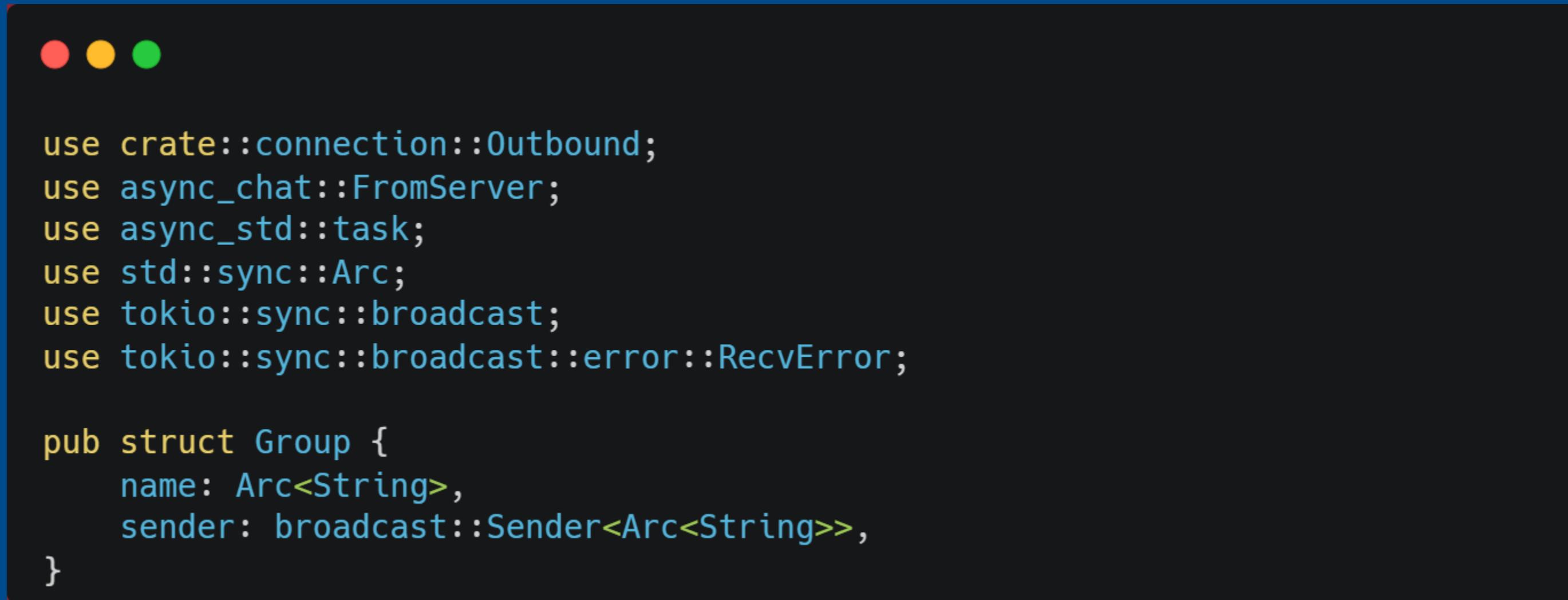
- 서버에서 `group::Group` 타입은 채팅 그룹을 표현한다.
 - 이 타입은 `connection::serve`가 호출하는 두 가지 메소드만 지원하면 되는데, 새 멤버를 추가하는 `join`과 메시지를 보내는 `post`가 바로 그것이다. 보낸 메시지는 모든 멤버에게 전달되어야 한다.
- 바로 배압(Backpressure)을 잘 처리해야 하는 부분이 바로 여기다.
여기에는 요구 사항 몇 가지가 서로 팽팽하게 맞서는 모양새로 돼 있다.
 - 어떤 멤버가 (네트워크 연결이 느리다던지 하는 등의 이유로) 그룹에 전달된 메시지를 그때그때 따라잡을 수 없더라도 그룹 안에 있는 다른 멤버가 영향을 받아서는 안 된다.
 - 비록 좀 뒤쳐지는 멤버가 있더라도 어떻게든 대화에 다시 합류해서 계속 참여할 방법이 있어야 한다.
 - 메시지를 버퍼링하는 데 쓰는 메모리가 한도 없이 늘어나면 안 된다.

- 이런 요구는 다대다 통신 패턴을 구현할 때면 늘 있는 일이다.
 - 그래서 tokio 크레이트는 합리적인 절충안 하나를 구현하고 있는 브로드캐스트 채널(Broadcast Channel) 타입을 제공한다. tokio 브로드캐스트 채널은 서로 다른 여러 스레드가 태스크가 값을 주고 받을 수 있는 값(채팅 메시지)의 큐다.
 - '브로드캐스트' 채널이라고 부르는 이유는 모든 소비자가 전달된 개별 값의 복사본을 받기 때문이다.
(이로 인해 값 타입은 반드시 Clone을 구현해야 된다.)
 - 보통 브로드캐스트 채널은 모든 소비자가 복사본을 받을 때까지 메시지를 큐에 유지해 둔다.
그러나 큐의 길이가 채널을 생성할 때 설정한 최대 용량을 넘어서면 가장 오래된 메시지가 삭제된다.
따라서 이를 따라잡지 못한 소비자는 다음 메시지를 가져오려고 할 때 오류를 받게 되며,
채널은 해당 소비자가 다음번에 현재 시점에서 가장 오래된 메시지를 받아볼 수 있도록 조치한다.

- 다음은 최대 16개의 값을 가질 수 있는 브로드캐스트 채널을 보여 준다.
 - 두 Sender는 메시지를 큐에 넣고 네 Receiver는 메시지를 큐에서 뺀다. (정확하게는 메시지를 큐에서 복사한다.)
 - Receiver B는 14개의 메시지를 덜 받았고, Receiver C는 7개의 메시지를 덜 받았고, Receiver D는 메시지를 전부 받았다. Receiver A는 완전히 뒤쳐져서 아직 받지도 못한 11개의 메시지가 삭제된 상태다.
 - 따라서 다음번에 메시지를 가져오려고 하면 상황 설명이 담긴 오류가 반환되며 실패하고, 다음번에 현재 시점에서 가장 오래된 메시지를 받아볼 수 있도록 조치한다.



- 채팅 서버는 각 채팅 그룹을 `Arc<String>` 값을 실어 나르는 브로드캐스트 채널로 표현한다.
그룹에 메시지를 보내면 현재 시점의 모든 멤버에게 브로드캐스트한다.



The screenshot shows a terminal window with three colored dots (red, yellow, green) at the top left. The terminal displays the following Rust code:

```
use crate::connection::Outbound;
use async_chat::FromServer;
use async_std::task;
use std::sync::Arc;
use tokio::sync::broadcast;
use tokio::sync::broadcast::error::RecvError;

pub struct Group {
    name: Arc<String>,
    sender: broadcast::Sender<Arc<String>>,
}
```

채팅 그룹

SNU SCSC + WaffleStudio 스터디
Week 12: Asynchronous Programming, Part 3

- 채팅 서버는 각 채팅 그룹을 `Arc<String>` 값을 실어 나르는 브로드캐스트 채널로 표현한다.
그룹에 메시지를 보내면 현재 시점의 모든 멤버에게 브로드캐스트한다.



```
impl Group {
    pub fn new(name: Arc<String>) -> Group {
        let (sender, _receiver) = broadcast::channel(1000);
        Group { name, sender }
    }

    pub fn join(&self, outbound: Arc<Outbound>) {
        let receiver = self.sender.subscribe();

        task::spawn(handle_subscriber(self.name.clone(), receiver, outbound));
    }

    pub fn post(&self, message: Arc<String>) {
        // This only returns an error when there are no subscribers.
        // A connection's outgoing side can exit, dropping its subscription,
        // slightly before its incoming side, which may end up trying to send
        // a message to an empty group.
        let _ignored = self.sender.send(message);
    }
}
```

- **코드 설명**
 - Group 구조체에는 채팅 그룹의 이름과 더불어 그 그룹의 브로드캐스트 채널에서 Sender를 표현하는 `broadcast::Sender`가 들어 있다.
 - `new` 함수는 `broadcast::channel`을 호출해서 최대 1,000개의 메시지를 가질 수 있는 브로드캐스트 채널을 만든다.
 - `channel` 함수는 Sender와 Receiver를 모두 반환하지만 그룹이 아직 멤버를 갖지 않으므로 현재 시점에서 Receiver는 필요 없다.
 - 새 그룹을 그룹에 추가하기 위해서 `join` 메소드는 Sender의 `subscribe` 메소드를 호출해 해당 채널의 Receiver를 새로 하나 만든다. 그런 다음 Receiver의 메시지를 모니터링해서 이를 다시 클라이언트로 보내는 `handle_subscribe` 함수를 비동기 태스크로 생성해 실행한다.

- 코드 설명
 - 이런 세부 사항을 알고 나면 post 메소드가 하는 일을 쉽게 이해할 수 있는데,
이 메소드는 단순히 메시지를 브로드캐스트 채널에 보낸다.
 - 채널이 실어 나르는 값은 `Arc<String>` 값이므로 Receiver마다 메시지의 복사본을 넘겨주더라도
그 메시지의 레퍼런스 카운트만 증가할 뿐 복사나 힙 할당이 전혀 발생하지 않는다.
 - 모든 구독자가 메시지를 전송하고 나면 레퍼런스 카운트는 0으로 떨어지고 메시지는 해제된다.

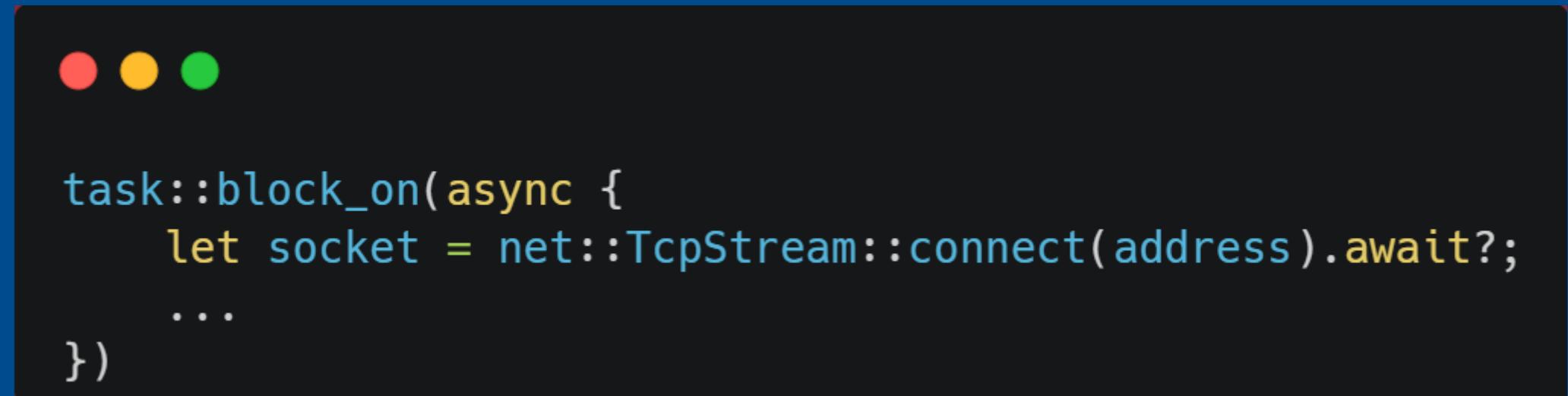
- `handle_subscriber`의 정의는 다음과 같다.

```
● ● ●  
  
async fn handle_subscriber(  
    group_name: Arc<String>,  
    mut receiver: broadcast::Receiver<Arc<String>>,  
    outbound: Arc<Outbound>,  
) {  
    loop {  
        let packet = match receiver.recv().await {  
            Ok(message) => FromServer::Message {  
                group_name: group_name.clone(),  
                message: message.clone(),  
            },  
            Err(RecvError::Lagged(n)) => {  
                FromServer::Error(format!("Dropped {} messages from {}", group_name, n))  
            }  
            Err(RecvError::Closed) => break,  
        };  
  
        if outbound.send(packet).await.is_err() {  
            break;  
        }  
    }  
}
```

- **코드 설명**
 - 브로드캐스트 채널에서 메시지를 받은 후 이를 다시 고유된 Outbound 값을 통해서 클라이언트로 전송하는 반복문이다.
반복문이 브로드캐스트 채널을 그때그때 따라잡지 못해서 Lagged 오류를 받게 되면
이 사실을 클라이언트에 있는 그대로 알린다.
 - 연결이 끊겼든지 하는 등의 이유로 패킷을 다시 클라이언트로 보내는데 완전히 실패하면
`handle_subscriber`가 반복문을 빠져나와 복귀하므로 비동기 태스크는 종료된다.
그리고 브로드캐스트 채널의 `Receiver`가 드롭되어 채널 구독이 해제된다.
이런 식으로 연결이 끊어지면 다음번에 그룹이 메시지를 보내려 할 때 해당 그룹의 멤버 등록이 취소된다.
 - 단, 채팅 그룹은 그룹 테이블에서 제거하지 않기 때문에 폐쇄되지 않는다.
그러나 완성도를 위해 `handle_subscriber`는 태스크가 종료할 때 발생하는 `Closed` 오류를 처리하도록 되어 있다.

- 채팅 서버 분석
 - TcpListener와 broadcast 채널 같은 비동기 기본 요소를 써서 코드를 작성하는 법과 block_on과 spawn 같은 이그제큐터를 써서 실행을 끌어나가는 법을 보여준다.
이제 이런 것들이 어떤 식으로 구현되어 있는지를 들여다 보자.
 - 핵심은 퓨처가 Poll::Pending을 반환할 때 이그제큐터를 조율해서 적절한 시점에 다시 폴링하게 만드는 방식에 있다.

- 채팅 클라이언트의 `main` 함수에서 다음 코드를 실행할 때 무슨 일이 벌어지는지 생각해 보자.
 - `block_on`이 맨 처음 `async` 블록의 퓨처를 폴링할 때는 네트워크 연결이 바로 준비되지 못할 게 뻔하므로 `block_on`은 잠자기 상태에 들어간다. 그럼 언제 깨어나야 할까? 어떤 식으로든 네트워크 연결이 준비되면 `TcpStream`이 `block_on`에게 `async` 블록의 퓨처를 다시 폴링해야 한다고 알려줘야 한다. 왜냐하면 이번에 다시 폴링하면 `await`가 완료되고, `async` 블록이 진도를 빨 수 있다는 걸 알고 있는게 바로 `TcpStream`이기 때문이다.



```
task::block_on(async {
    let socket = net::TcpStream::connect(address).await?;
    ...
})
```

- 채팅 클라이언트의 `main` 함수에서 다음 코드를 실행할 때 무슨 일이 벌어지는지 생각해 보자.
 - `block_on` 같은 이그제큐터는 퓨처를 폴링할 때 웨이커(Waker)라고 하는 콜백을 꼭 넘겨야 한다. `Future` 트레잇의 규칙에 따르면 퓨처는 준비 상태가 아닐 때 일단 `Poll::Pending`을 반환하고 나중에 퓨처를 다시 폴링해도 좋은 시점이 되면 웨이커를 호출하도록 조율해야 한다.



A screenshot of a terminal window with three colored window control buttons (red, yellow, green) at the top. The terminal displays the following C++ code:

```
task::block_on(async {
    let socket = net::TcpStream::connect(address).await?;
    ...
})
```

- 따라서 손으로 쓴 Future의 구현은 다음과 같은 모습일 때가 많다.

```
use std::task::Waker;

struct MyPrimitiveFuture {
    ...
    waker: Option<Waker>,
}

impl Future for MyPrimitiveFuture {
    type Output = ...;

    fn poll(mut self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<...> {
        ...

        if ... future is ready ... {
            return Poll::Ready(final_value);
        }

        // Save the waker for later.
        self.waker = Some(cx.waker().clone());

        Poll::Pending
    }
}
```

- **코드 설명**

- 즉, 퓨처의 값이 준비됐으면 이를 반환하고,
그렇지 않으면 Context에 있는 웨이커의 복사본을 어딘가 넣어 두고 Poll::Pending을 반환한다.
- 퓨처를 다시 폴링해도 좋은 시점이 되면 퓨처는 웨이커의 wake 메소드를 호출해서
자신을 폴링했던 마지막 이그제큐터에게 이 사실을 알려야 한다.



```
// If we have a waker, invoke it, and clear `self.waker`.
if let Some(waker) = self.waker.take() {
    waker.wake();
}
```

- 이상적으로는 이그제큐터와 퓨처가 번갈아가며 폴링하고 깨어나기를 반복하는게 바람직하다.
이그제큐터가 퓨처를 폴링하다가 잠자기 상태에 들어가면, 퓨처가 웨이커를 호출해서 이그제큐터를 깨우고,
그럼 다시 이그제큐터가 퓨처를 폴링하다가 잠자기 상태에 들어가고 하는 식으로 말이다.

- 코드 설명
 - `async` 함수와 블록의 퓨처가 웨이커 자체를 다루는 건 아니다.
단지 주어진 컨텍스트를 자신이 기다리는 서브퓨처에 넘겨서 웨이커를 저장하고 호출하는 의무를 위임할 뿐이다.
채팅 클라이언트에서는 `async` 블록의 퓨처에 대한 첫 번째 폴링이 `TcpStream::connect`의 퓨처를 기다릴 때
컨텍스트를 넘긴다. 이어지는 폴링도 마찬가지로 블록이 기다리는 다음 차례의 퓨처에 자신의 컨텍스트를 넘긴다.
 - `TcpStream::connect`의 퓨처 핸들은 앞서 본 예처럼 폴링된다.
즉, 연결이 준비되길 기다렸다가 웨이커를 호출하는 도우미 스레드에 웨이커를 전달한다.

- **코드 설명**

- Waker는 Clone과 Send를 구현하고 있으므로 퓨처는 언제라도 웨이커의 복사본을 만들어 필요할 때 다른 스레드로 보낼 수 있다. Waker::wake는 웨이커를 소비한다.
그렇지 않은 wake_by_ref 메소드도 있지만 일부 이그제큐터는 소비하는 버전으로 좀 더 효율적으로 구현할 수 있다.
- 이그제큐터가 퓨처를 과도하게 폴링하는 건 전혀 문제될 게 없으며 단지 비효율적일 뿐이다.
하지만 퓨처는 폴링이 실제로 진도를 빨 수 있을 때만 웨이커를 호출하도록 주의해야 한다.
그렇지 않은데 깨워서 폴링하는 주기가 생기면 이그제큐터가 잠자기 상태에 들어가는 걸 방해해서 전력을 낭비하게 되고 프로세서가 다른 태스크에 쏟아야 할 시간을 확보하기 어렵게 된다.

- <https://www.rust-lang.org/>
- <https://doc.rust-lang.org/book/>
- <https://tourofrust.com/>
- The Rust Programming Language (No Starch Press, 2019)
- Programming Rust, 2nd Edition (O'Reilly, 2021)

Thank you!