.UBA fiuba FACULTAD DE INGENIERÍA

Taller de Programación I

Introducción a Redes y Sockets en Rust

Ing. Pablo A. Deymonnaz
pdeymon@fi.uba.ar

Facultad de Ingeniería
Universidad de Buenos Aires

Índice

1. Introducción

2. Sockets en Rust

.UBAfiuba FACULTAD DE INGENIERÍA

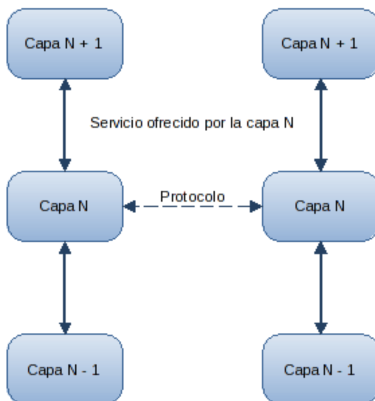
.UBA fiuba FACULTAD DE INGENIERÍA

Introducción

- Permiten la comunicación entre dos procesos diferentes
  - En la misma máquina
  - En dos máquinas diferentes
- Se usan en aplicaciones que implementan el modelo cliente - servidor:
  - Cliente: es activo porque inicia la interacción con el servidor
  - Servidor: es pasivo porque espera recibir las peticiones de los clientes

Modelo de capas del software de red

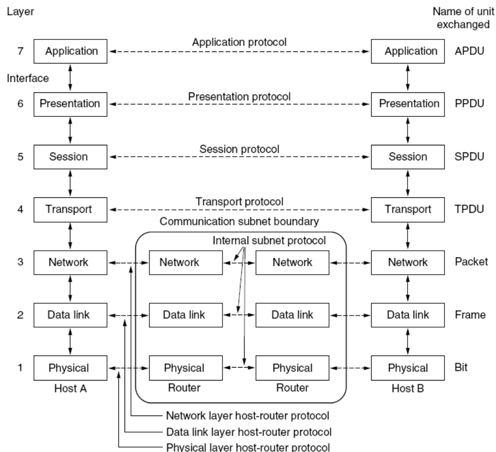
Modelo de capas



Tipos de servicio

- Sin conexión
  - Los datos se envían al receptor y no hay control de flujo ni de errores.
- Sin conexión con ACK
  - Por cada dato recibido, el receptor envía un acuse de recibo conocido como *ACK*.
- Con conexión
  - Tres fases: establecimiento de la conexión, intercambio de datos y cierre de la conexión. Hay control de flujo y control de errores.

Modelo OSI



Tipos de Sockets

- *Stream sockets:* usan el protocolo TCP: entrega garantizada del flujo de bytes.
- *Datagram sockets:* usan el protocolo UDP: la entrega no está garantizada; servicio sin conexión.
- *Raw sockets:* permiten a las aplicaciones enviar paquetes IP.
- *Sequenced packet sockets:* similares a *stream sockets*, pero preservan los delimitadores de registro. Utilizan el protocolo SPP (Sequenced Packet Protocol).

Índice

1. Introducción

2. Sockets en Rust
   - Servidor
   - Cliente
   - Finalización

Servidor TCP

La biblioteca de networking de Rust se encuentra en el módulo **std::net**.
Para el **Servidor**.

- **Primer Paso:** Asociar el socket a una dirección.
  El método **bind** crea un nuevo *TcpListener* y lo asocia a una dirección específica.

```
pub fn bind<A: ToSocketAddrs>(addr: A) -> Result<TcpListener>
```

El *listener* retornado está listo para aceptar conexiones.

```
let listener = TcpListener::bind("127.0.0.1:80")?;
```

UBA fiuba FACULTAD DE INGENIERÍA

Servidor TCP

- **Segundo Paso:** Obtener conexiones establecidas.

Sobre la estructura *TcpListener* se obtienen conexiones establecidas. El método **incoming** retorna un iterador que devuelve una secuencia de streams de tipo *TcpStream*.

```
pub fn incoming(&self) -> Incoming<'_>
```

Cada stream representa una conexión abierta entre el cliente y el servidor.

```
for stream in listener.incoming() {
  let stream = stream.unwrap();
  println!("Conexion establecida!");
}
```

La iteración es sobre “intentos de conexiones”. Puede retornar *Err*.

Servidor TCP

- **Segundo Paso (forma alternativa):** Obtener conexiones establecidas con **accept**.

EL método **accept** obtiene una conexión establecida de un listener.

```
pub fn accept(&self) -> Result<(TcpStream, SocketAddr)>
```

El hilo se bloquea hasta que haya una conexión establecida.

```
match listener.accept() {
  Ok((_socket, addr)) => println!("nuevo cliente: {:?}", addr),
  Err(e) => println!("error: {:?}", e),
}
```

Servidor TCP

- **Tercer Paso:** Leer datos del socket: **read**.

TcpStream implementa el método **read** (del trait *std::io::Read*).

```
fn read(&mut self, buf: &mut [u8]) -> Result<usize>
```

por ejemplo:

```
let mut buffer = [0; 1024];
stream.read(&mut buffer).unwrap();
```

Servidor TCP

- **Escribir una respuesta:** El servidor envía una respuesta a una petición del cliente.

TcpStream implementa el método **write** (del trait *std::io::Write*).

```
fn write(&mut self, buf: &[u8]) -> Result<usize>
```

por ejemplo:

```
let response = "Respuesta!\n";

stream.write(response.as_bytes()).unwrap();
stream.flush().unwrap();
```

El método *flush* realiza una espera, previniendo que el programa continúe sin haber escrito en la conexión todos los bytes.
Flush this output stream, ensuring that all intermediately buffered contents reach their destination.

Cliente TCP

El cliente debe establecer la **conexión** con el servidor.

Construir la dirección de destino

- **A partir de una dirección IP:**

```
use std::net::{IpAddr, Ipv4Addr, SocketAddr};
let socket =
  SocketAddr::new(IpAddr::V4(Ipv4Addr::new(127, 0, 0, 1)), 8080)
```

- **A partir de un nombre:** método **to_socket_addrs**

```
fn to_socket_addrs(&self) -> Result<Self::Iter>
```

por ejemplo:

```
let mut addrs_iter = "localhost:443".to_socket_addrs().unwrap();
```

Devuelve un iterador de direcciones.

Cliente TCP

Conectarse al servidor: el cliente ejecuta el método **connect**

```
pub fn connect<A: ToSocketAddrs>(addr: A) -> Result<TcpStream>
```

Este método abre una conexión al host remoto.
Si se le envía un array de direcciones, intenta conectarse a cada una, hasta lograrlo.

```
let addrs = [
  SocketAddr::from(([127, 0, 0, 1], 8080)),
  SocketAddr::from(([127, 0, 0, 1], 8081)),
];
if let Ok(stream) = TcpStream::connect(&addrs[..]) {
  println!("Connectado al servidor!");
} else {
  println!("No se pudo conectar...");
}
```

Cliente TCP

Para enviar y recibir datos, el cliente ejecuta los métodos **read** y **write** igual que el servidor.

Finalizar conexión

- El cierre de la conexión TCP puede ser realizado de forma individual.
- La conexión establecida con *TcpStream* se cierra cuando el valor ejecuta *drop*. Esto inicia el envío del mensaje *close* de TCP.
- El método shutdown pude cerrar el extremo de escritura, de lectura o ambos.

```
pub fn shutdown(&self, how: Shutdown) -> Result<()>
```

UBAfiuba FACULTAD DE INGENIERÍA

Bibliografía

- **The Rust Programming Language**, *https://doc.rust-lang.org/book/*
  - Chapter 20. *Final Project: Building a Multithreaded Web Server*
- **Módulo std::net**, *https://doc.rust-lang.org/std/net/*
- **Computer Networks**, Andrew S. Tanenbaum, cuarta edición.
- **Unix Network Programming - Volume 1 - The Sockets Networking API**, Richard Stevens, tercera edición