FERNANDO ELIAS BEVANS CATAÑO
ING. SOFTWARE 2025

Resumen del Funcionamiento del Proyecto

El proyecto es un sistema de comunicación en red que actúa como un **puente de datos**. El servidor de **Rust** recibe comandos de control de un cliente de **Python**, confirma la recepción y luego, como intermediario, retransmite el estado completo de los servos a un tercer dispositivo.

El flujo es el siguiente:

1. El cliente de **Python** (192.168.0.81) se conecta al servidor de **Rust** (192.168.0.20:9090).
2. Python envía un comando **NMEA** para un servo (ej., Ronza).
3. Rust recibe el comando, lo procesa y:
    - **Paso 1: Confirmación.** Envía un mensaje $ACK de vuelta a Python para confirmarle que la trama fue recibida correctamente.
    - **Paso 2: Retransmisión.** Se conecta a un segundo cliente (192.168.0.249:9090), toma los últimos valores conocidos de *ambos* servos (Ronza y Elevación), y envía una única trama NMEA que contiene el estado completo de los dos.

De esta manera, la aplicación de Python se asegura de que el comando llegó a su destino, mientras que el cliente 192.168.0.249 siempre recibe la información actualizada y combinada de los dos servos en una sola trama.

Código de Rust con Comentarios

Este es el servidor que hace todo el trabajo de recepción, retransmisión y gestión del estado.

Rust

use std::net::{TcpListener, TcpStream}; // 💻 Módulos para crear y manejar conexiones de red TCP.

use std::io::{self, Read, Write}; // 📥 Módulos para leer y escribir datos de la red.

use std::str; // 📝 Módulo para convertir bytes a cadenas de texto (strings).

use std::thread; // ⚙️ Módulo para crear hilos, permitiendo manejar múltiples clientes a la vez.

```
use std::collections::HashMap; // 🗺️ Estructura de datos para guardar los valores de los servos por su nombre.
use std::sync::Mutex; // 🔒 Mecanismo para proteger el acceso a los datos compartidos entre hilos.

// lazy_static permite inicializar variables estáticas globales de forma segura.
// Esto es crucial para que todos los hilos puedan acceder al mismo HashMap.
lazy_static::lazy_static! {
    static ref SERVO_VALUES: Mutex<HashMap<String, String>> = {
        let mut map = HashMap::new();
        map.insert("RONZA".to_string(), "0".to_string()); // ➡️ Inicializa el valor de Ronza.
        map.insert("ELEVACION".to_string(), "0".to_string()); // ➡️ Inicializa el valor de Elevación.
        Mutex::new(map) // 🔒 Protege el mapa con un Mutex.
    };
}

// Función para calcular el checksum NMEA (XOR de todos los bytes).
fn calculate_nmea_checksum(data: &str) -> u8 {
    let mut checksum: u8 = 0;
    for byte in data.bytes() {
        checksum ^= byte; // 🔄 Realiza la operación XOR para cada byte de la cadena.
    }
    checksum // ➡️ Retorna el checksum final.
}

// Función que se ejecuta para cada cliente conectado.
fn handle_client(mut stream: TcpStream) {
    println!("Cliente conectado desde: {}", stream.peer_addr().unwrap()); // 🗣️ Imprime la dirección IP del cliente.
    let mut buffer = [0; 256]; // 📦 Un buffer para guardar los datos recibidos.
```

```
    loop { // 🔄 Bucle para leer datos continuamente del cliente.
        match stream.read(&mut buffer) {
            Ok(0) => {
                println!("Cliente desconectado: {}", stream.peer_addr().unwrap());
                return; // 🚪 Sale del hilo si el cliente se desconecta.
            }
            Ok(bytes_read) => {
                let received = str::from_utf8(&buffer[..bytes_read])
                    .unwrap_or_default()
                    .trim_matches(char::from(0))
                    .trim(); // 🧹 Convierte bytes a string y limpia caracteres no deseados.

                println!("Recibido: {}", received);

                // Valida que la trama NMEA tenga el formato correcto.
                if received.starts_with("$") && received.contains("*") {
                    let parts: Vec<&str> = received.split('*').collect();
                    if parts.len() == 2 {
                        let data = parts[0].trim_start_matches("$");
                        let data_parts: Vec<&str> = data.split(',').collect(); // ✂️ Divide la trama por las
comas.

                        if data_parts.len() >= 2 {
                            let servo_type = data_parts[1]; // "RONZA" o "ELEVACION"
                            let servo_value = data_parts.get(2).unwrap_or(&"0"); // Valor de la posición.

                            // Adquiere el bloqueo del Mutex para actualizar el estado compartido.
                            let mut values = SERVO_VALUES.lock().unwrap();
                            if values.contains_key(servo_type) {
```

```
                values.insert(servo_type.to_string(), servo_value.to_string()); // 💾 Guarda el
nuevo valor.
            }
            drop(values); // 🔓 Libera el bloqueo del Mutex.

            // 1. Envía el ACK de vuelta al cliente de Python
            let response_to_python = format!("$ACK,{}*{:02X}\r\n", servo_type, 0);
            match stream.write_all(response_to_python.as_bytes()) {
                Ok(_) => println!("Enviado ACK a Python: {}", response_to_python.trim()),
                Err(e) => {
                    eprintln!("Error al enviar ACK: {}", e);
                    return;
                }
            }

            // 2. Conectarse y enviar la trama combinada al segundo cliente
            let destination_addr = "192.168.0.249:9090";
            match TcpStream::connect(destination_addr) {
                Ok(mut dest_stream) => {
                    // Adquiere un nuevo bloqueo para leer los valores de los servos.
                    let values = SERVO_VALUES.lock().unwrap();
                    let ronza_val = values.get("RONZA").unwrap();
                    let elevacion_val = values.get("ELEVACION").unwrap();

                    // Construye la nueva trama NMEA combinada.
                    let nmea_data = format!("ACK,{},{}", ronza_val, elevacion_val); // ➡
"ACK,110,90"
                    let checksum = calculate_nmea_checksum(&nmea_data); // ➡ Calcula el
checksum.
                    let full_nmea_frame = format!("${}*{:02X}\r\n", nmea_data, checksum); // ➡
Ensambla la trama completa.
```

```
                            // Envía la trama al segundo cliente.
                            match dest_stream.write_all(full_nmea_frame.as_bytes()) {
                                Ok(_) => println!("Trama NMEA enviada a {}: {}", destination_addr,
full_nmea_frame.trim()),
                                Err(e) => eprintln!("Error al enviar datos a {}: {}", destination_addr, e),
                            }
                        }
                        Err(e) => {
                            eprintln!("Error al conectar con {}: {}", destination_addr, e);
                        }
                    }
                }
            }
        } else {
            eprintln!("Trama NMEA no válida o incompleta: {}", received);
        }
    }
    Err(e) => {
        eprintln!("Error de lectura: {}", e);
        return;
    }
  }
 }
}

fn main() -> io::Result<()> {
    let listen_addr = "192.168.0.20:9090";
    let listener = TcpListener::bind(listen_addr)?; //  Comienza a escuchar en el puerto 9090.
```

```
    println!("Servidor Rust escuchando en {}", listen_addr);

    // Bucle para aceptar nuevas conexiones.
    for stream in listener.incoming() {
        match stream {
            Ok(stream) => {
                thread::spawn(|| handle_client(stream)); // 🚀 Lanza un nuevo hilo para cada conexión.
            }
            Err(e) => {
                eprintln!("Error al aceptar la conexión: {}", e);
            }
        }
    }

    Ok(())
}
```

Código de Python con Comentarios

Este es un ejemplo simplificado de cómo el cliente de Python enviaría un comando al servidor de Rust.

Python

```
import socket
import time

# Función para calcular el checksum de una trama NMEA en Python
def calculate_nmea_checksum(data):
    checksum = 0
    for byte in data.encode('ascii'):
        checksum ^= byte
```

```
    return '{:02X}'.format(checksum)

def send_command(servo, value):
    host = '192.168.0.20' # 📍 Dirección del servidor de Rust
    port = 9090 # 🔌 Puerto del servidor de Rust

    # Trama NMEA que se va a enviar
    command_data = f'SERVOS,{servo},{value}'
    checksum = calculate_nmea_checksum(command_data)
    nmea_frame = f'${command_data}*{checksum}\r\n'

    try:
        # Crea un socket TCP/IP
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
            s.connect((host, port)) # 🤝 Se conecta al servidor.
            print(f"Conectado a {host}:{port}")

            s.sendall(nmea_frame.encode('ascii')) # 🚀 Envía la trama NMEA.
            print(f"Enviado: {nmea_frame.strip()}")

            # Espera la respuesta (ACK) del servidor de Rust
            ack_response = s.recv(1024) # 📥 Recibe hasta 1024 bytes.
            print(f"ACK recibido: {ack_response.decode('ascii').strip()}")

    except ConnectionRefusedError:
        print("Error: Conexión rechazada. Asegúrate de que el servidor Rust esté en ejecución.")
    except Exception as e:
```

```
        print(f"Ocurrió un error: {e}")


if __name__ == "__main__":
    send_command("RONZA", 110) # ➡ Envía el primer comando.
    time.sleep(2) # Pausa de 2 segundos.
    send_command("ELEVACION", 130) # ➡ Envía el segundo comando.
```

Resumen del Código de Python

Este script crea una GUI interactiva con Tkinter para que el usuario pueda conectarse a un servidor de Rust, enviar comandos para controlar la posición de dos servos (Ronza y Elevación), y ver las respuestas en tiempo real. El código está diseñado para ser **robusto y tolerante a fallos** mediante el uso de **hilos**, **colas** y **timeouts**, lo que evita que la interfaz se congele.

Explicación del Código Línea por Línea

1. Módulos y Configuración

- import tkinter as tk, ttk, messagebox, scrolledtext: Módulos de Python para crear la interfaz gráfica (botones, campos de texto, etc.) y mostrar mensajes emergentes.
- import socket, threading, queue, time, sys: Módulos para la comunicación de red, la ejecución de tareas en paralelo, la gestión de colas de mensajes y el manejo de tiempo y errores.
- HOST_IP, HOST_PORT: Variables globales para la dirección del servidor de Rust.
- sock, ethernet_read_thread, read_thread_running: Variables globales para la conexión y el hilo de lectura del socket.
- ethernet_queue: Una **cola** para transferir datos de forma segura entre el hilo de lectura y el hilo principal de la GUI, evitando errores de concurrencia.

- move_active, is_moving, last_sent_time: Diccionarios y variables booleanas que actúan como **banderas de estado**. Controlan si un servo está en movimiento (is_moving), si un botón de movimiento continuo está presionado (move_active) y el tiempo del último comando enviado.
- TIMEOUT_SECONDS, lock: El **lock** es un **threading.Lock** que protege las banderas de estado para que no haya conflictos cuando son accedidas por diferentes hilos.

2. Funciones de Conexión y Desconexión

- conectar_ethernet():
    - Lee la IP y el puerto de la interfaz gráfica.
    - Crea un socket TCP/IP y usa un try-except para intentar la conexión.
    - Si la conexión es exitosa, **habilita y deshabilita botones** para reflejar el estado, y más importante aún, **inicia un nuevo hilo** (read_from_ethernet) para leer datos del socket en segundo plano.
- desconectar_ethernet():
    - Detiene el hilo de lectura del socket estableciendo la bandera read_thread_running en False.
    - Cierra el socket, actualiza la interfaz y limpia el estado.

3. Hilo y Cola de Mensajes

- read_from_ethernet():
    - Esta función se ejecuta en un **hilo separado**.
    - Usa un bucle while para leer datos del socket de forma continua.
    - Cuando recibe datos, los procesa para encontrar mensajes completos (terminados en \n).
    - Los mensajes completos se colocan en la ethernet_queue para que el hilo principal los procese, asegurando que la lectura no bloquee la interfaz gráfica.
- update_ethernet_terminal():
    - Se ejecuta periódicamente en el hilo principal de la GUI.

- Lee los mensajes de la ethernet_queue.
- **Procesa los mensajes**: si la línea recibida es un ACK del servidor de Rust, **resetea la bandera is_moving** para el servo correspondiente. Esto indica que el servo está libre para recibir un nuevo comando.
- Actualiza el scrolledtext (la terminal) con los mensajes recibidos.

4. Control de Movimiento

- check_timeouts():
  - Esta función verifica periódicamente si algún comando enviado ha excedido el TIMEOUT_SECONDS sin recibir un ACK del servidor.
  - Si detecta un timeout, **resetea la bandera is_moving** y muestra una advertencia en la terminal, evitando que la aplicación se bloquee indefinidamente si se pierde un mensaje de confirmación.
- generar_trama_nmea():
  - Construye la trama NMEA completa a partir del tipo de comando, el eje y el valor.
  - **Calcula el checksum** usando la operación XOR, asegurando la integridad de la trama.
- enviar_datos_ethernet():
  - Envía la trama NMEA al servidor de Rust.
  - Inmediatamente después de enviar, **activa la bandera is_moving** y registra el tiempo del envío. Esto previene que se envíen más comandos al mismo servo hasta que se reciba la confirmación o se produzca un timeout.
- control_paso(), start_continuous_move(), perform_continuous_move(), stop_continuous_move():
  - Manejan el movimiento del servo.
  - control_paso() envía un solo comando.
  - Las funciones start_, perform_ y stop_ se encargan del movimiento continuo, enviando comandos repetidamente siempre y cuando la bandera is_moving esté False (es decir, cuando el servo esté libre).

- enviar_posicion_especifica():
    - Envía un comando para mover un servo a una posición exacta ingresada por el usuario.
- set_speed():
    - Envía un comando para controlar la velocidad de movimiento del servo, actualizando la etiqueta en la GUI para mostrar la velocidad actual.

Resumen Final

El código de Python es una interfaz de control robusta que se comunica con el servidor de Rust. La clave de su fiabilidad reside en la **separación de responsabilidades**:

- **Hilo de la GUI:** Maneja la interacción del usuario y la actualización de la pantalla.
- **Hilo de Lectura:** Se dedica exclusivamente a escuchar la red sin bloquear la GUI.
- **Colas y Banderas:** Permiten una comunicación segura entre los hilos y un control preciso sobre el estado de cada servo, previniendo el envío de comandos no deseados y gestionando la recuperación de errores de comunicación.