Implementación de Servidor HTTP

Erick Kauffmann, Luany Masís, José Sequeira
Instituto Tecnológico de Costa Rica
Sede Central Cartago

***Abstract*—Este proyecto consiste en el diseño e implementación de un servidor HTTP/1.0 concurrente desarrollado en Rust, orientado a la validación y prueba automatizada de algoritmos y procesos. Se implementan mecanismos robustos para el manejo de múltiples conexiones simultáneas, escalabilidad horizontal, y procesamiento modular de comandos específicos, asegurando una arquitectura segura, eficiente y mantenible.**

***Index Terms*—Servidor HTTP, concurrencia, Rust, sockets, pruebas automatizadas, escalabilidad, sistemas operativos.**

I. Introducción

En el contexto de la asignatura de Principios de Sistemas Operativos, este proyecto busca desarrollar un servidor HTTP funcional que soporte múltiples clientes concurrentes, basado exclusivamente en librerías estándar de Rust para sockets y concurrencia. El objetivo es fomentar competencias en la gestión segura de recursos, diseño modular y pruebas exhaustivas, contribuyendo a la formación práctica en sistemas distribuidos y redes.

II. Marco Teórico

A. Fundamentos sobre servidores HTTP

El protocolo HTTP (Hypertext Transfer Protocol) es un estándar para la comunicación entre clientes y servidores en la web. HTTP/1.0, especificado en RFC 1945, establece un modelo simple basado en conexiones TCP donde el cliente envía una solicitud y el servidor responde con el recurso solicitado o el resultado de una operación. HTTP utiliza métodos como GET, POST y responde con códigos de estado que indican el éxito o error de la operación. La ausencia de cifrado en HTTP/1.0 implica que los datos viajan en texto plano, lo que limita su uso en contextos donde la seguridad sea crítica.

B. Manejo de procesos

En sistemas operativos Unix-like, el manejo de procesos es fundamental para ejecutar múltiples tareas simultáneamente. En Rust, se emplean hilos (`threads`) para manejar la concurrencia de manera eficiente. La creación de hilos se realiza a través de la librería estándar `std::thread`, la cual permite generar unidades de ejecución concurrentes que pueden trabajar de forma independiente. La gestión adecuada de hilos, junto con mecanismos de sincronización como canales y Mutex para asegurar el acceso seguro a recursos compartidos, es crucial para evitar condiciones de carrera, bloqueos o el uso ineficiente de recursos. En este proyecto, el enfoque principal es la creación y manejo seguro de hilos, utilizando un modelo basado en pools de hilos, donde el trabajo se distribuye entre los hilos de manera eficiente a través de canales para su comunicación. Esta arquitectura asegura que el servidor pueda atender múltiples solicitudes de manera concurrente, sin bloquear el hilo principal.

C. Concurrencia

La concurrencia permite que múltiples tareas se ejecuten en solapamiento temporal, mejorando el rendimiento y la capacidad de respuesta del sistema. En Rust, la concurrencia se maneja de manera eficiente utilizando hilos seguros (`std::thread`) y mecanismos como canales (`std::sync::mpsc`) para la comunicación entre hilos y la sincronización de los datos compartidos. Se evita el uso indiscriminado de `sleep()` como mecanismo de sincronización, en favor de métodos más seguros y controlados, como el uso de Mutex y canales, los cuales garantizan que las tareas se distribuyan correctamente entre los hilos sin comprometer la seguridad de los datos. Además, en este proyecto se implementa un modelo de ejecución concurrente donde los trabajos se asignan a los hilos mediante un pool de hilos, lo que permite un manejo más eficiente de las solicitudes entrantes. El uso de async/await no es utilizado de manera predominante en este proyecto, ya que el enfoque principal se centra en la concurrencia basada en hilos y canales, optimizando el rendimiento de la aplicación en un entorno de tareas concurrentes sin la necesidad de utilizar programación asíncrona.

D. Sockets

Los sockets son un mecanismo de comunicación que permite intercambiar datos entre procesos, ya sea en la misma máquina o a través de redes. En el contexto TCP/IP, los sockets ofrecen una comunicación orientada a conexión confiable. La API de sockets en Rust (a través de `std::net`) permite crear servidores que escuchan en puertos específicos, aceptan conexiones y leen o escriben datos. El manejo eficiente de sockets es vital para la performance y estabilidad del servidor.

E. Arquitectura cliente-servidor

El modelo cliente-servidor es una arquitectura distribuida donde clientes envían solicitudes a servidores que procesan y responden. Este patrón separa responsabilidades, facilita escalabilidad y mantenimiento. En este proyecto, el servidor

actúa como un centro de procesamiento que recibe solicitudes HTTP, ejecuta comandos especializados y retorna resultados, soportando múltiples clientes de forma concurrente.

III. Diseño e Implementación

El servidor HTTP desarrollado para este proyecto sigue una arquitectura modular, diseñada desde cero exclusivamente con la biblioteca estándar de Rust. Su implementación se fundamenta en tres pilares técnicos: el manejo manual del protocolo HTTP/1.0, la concurrencia basada en un pool de hilos personalizados, y la organización modular de las funciones disponibles como comandos accesibles por rutas HTTP. Este enfoque permite un sistema escalable, seguro y fácilmente mantenible, cumpliendo con las restricciones del curso de no utilizar servidores embebidos ni bibliotecas de alto nivel.

A. Arquitectura General

El proyecto se organiza en cinco componentes principales:

- **Servidor principal (`main.rs`):** Es el punto de entrada del sistema. Configura el socket TCP, inicializa el thread pool y ejecuta el bucle principal que acepta conexiones. Cada conexión aceptada se convierte en una tarea que se envía al pool para ser atendida de manera concurrente.
- **Pool de hilos (`src/pool/`):** Implementa un pool personalizado con `std::thread`, canales MPSC y estructuras `Arc<Mutex<...>>`. Permite reutilizar hilos trabajadores evitando la sobrecarga del sistema.
- **Módulo del servidor (`src/server/`):** Encargado del parsing manual de solicitudes HTTP, el enrutamiento a comandos, y la construcción de respuestas válidas.
- **Funciones HTTP (`src/functions/`):** Cada comando accesible por ruta (e.g. `/fibonacci`) se implementa como módulo separado, permitiendo pruebas y mantenimiento independientes.
- **Manejo de errores (`src/errors/`):** Define errores tipados según su contexto (parsing, pool, servidor) y permite respuestas coherentes ante fallos.

B. Flujo de Ejecución

1) El servidor se lanza desde `main.rs` y escucha en un puerto definido.
2) Cada conexión TCP aceptada se encapsula como tarea y se envía al `ThreadPool`.
3) Un hilo trabajador toma la tarea, parsea la solicitud HTTP y la representa como una estructura `Request`.
4) La estructura se enruta mediante `routes.rs` y se ejecuta la función correspondiente.
5) Se genera una respuesta en formato HTTP/1.0 y se escribe de vuelta al cliente.

C. Rutas Implementadas

Cada una de las siguientes rutas se implementó como un módulo independiente:

- `/fibonacci?num=N`: Calcula el número N de la secuencia de Fibonacci.
- `/createfile?name=filename&content=text&repeat=x`: Crea un archivo con contenido repetido.
- `/deletefile?name=filename`: Elimina un archivo existente.
- `/reverse?text=abc`: Devuelve el texto invertido.
- `/toupper?text=abc`: Convierte texto a mayúsculas.
- `/random?count=n&min=a&max=b`: Genera números aleatorios.
- `/timestamp`: Retorna la hora actual en formato ISO-8601.
- `/hash?text=abc`: Devuelve el hash SHA-256 del texto.
- `/simulate?seconds=s&task=x`: Simula una tarea con retardo.
- `/sleep?seconds=s`: Introduce latencia artificial.
- `/loadtest?tasks=n&sleep=s`: Ejecuta múltiples tareas simuladas.
- `/status`: Reporta el estado del servidor (PID, uptime, hilos, conexiones).
- `/help`: Lista todos los comandos disponibles.

D. Manejo de Concurrencia

Rust provee mecanismos seguros de concurrencia. El proyecto utiliza:

- `std::thread` para crear hilos.
- `std::sync::mpsc` como canal para distribuir tareas.
- `Arc<Mutex<Receiver<...»` para compartir de forma segura el canal entre hilos.

Esto permite escalar el sistema sin condiciones de carrera, con una cola de tareas y procesamiento eficiente. No se usa `async/await`, respetando la especificación del curso.

E. Parsing y Respuesta HTTP

El servidor realiza parsing manual de las solicitudes HTTP:

- Se parsea la línea de inicio y parámetros.
- Se construye una estructura `Request`.
- Se genera una respuesta con `response.rs` incluyendo:
  - Código de estado (explicados más abajo).
  - Cabeceras como `Content-Type`, `Content-Length`, `Connection: close`.
  - Cuerpo con contenido o mensaje de error.

Códigos de Estado HTTP utilizados:

- **200 OK:** Solicitud válida, respuesta generada correctamente.
- **400 Bad Request:** Solicitud malformada o con parámetros inválidos.
- **404 Not Found:** Ruta o recurso no existe.
- **500 Internal Server Error:** Error inesperado en el servidor durante el procesamiento.

F. Robustez y Validación

El sistema devuelve respuestas coherentes ante errores. Los errores se tipifican y procesan según su contexto. El servidor

responde con códigos adecuados y mensajes interpretables. La salida del endpoint `/status` es en formato JSON para facilitar análisis por herramientas automatizadas.

G. Conclusión

Este diseño modular, concurrente y seguro representa una implementación profesional de un servidor HTTP/1.0 desde cero, alineado perfectamente con los objetivos del curso. Cumple con todos los requerimientos técnicos: sin bibliotecas externas, alto grado de separación de responsabilidades, y preparado para pruebas unitarias e integración.

IV. Estrategia de Pruebas

Para garantizar la calidad, robustez y cumplimiento funcional del servidor HTTP desarrollado, se diseñó una estrategia de pruebas integral que abarca diferentes niveles y técnicas. Estas pruebas se alinean con los requerimientos metodológicos definidos en el proyecto y tienen como objetivo detectar errores de forma temprana, validar los comportamientos esperados y garantizar la estabilidad bajo carga.

A. Pruebas Unitarias

Las pruebas unitarias se implementarán para cada uno de los módulos funcionales del servidor, en particular aquellos presentes en la carpeta `src/functions/`, como `fibonacci`, `hash`, `toupper`, entre otros.

Cada prueba validará los siguientes aspectos:

- Correcto funcionamiento con entradas válidas.
- Comportamiento esperado ante entradas inválidas o nulas.
- Manejo adecuado de errores y límites (edge cases).

Se utilizará el framework de pruebas de Rust (`cargo test`) y la herramienta `cargo tarpaulin` para evaluar la cobertura, buscando alcanzar al menos un 90% del código ejecutado por pruebas.

B. Pruebas de Integración

Estas pruebas se enfocan en la interacción entre los diferentes módulos del sistema:

- Flujo completo de procesamiento: desde que se recibe una solicitud HTTP hasta que se genera la respuesta.
- Verificación de la lógica de enrutamiento.
- Validación de la integración del pool de hilos con las tareas asignadas.

Se construirán pruebas utilizando herramientas como el Runner de Postman.

C. Pruebas Funcionales con Postman

Se diseñó una colección Postman con pruebas para cada una de las rutas implementadas. Esta colección incluye:

- Casos válidos que deben retornar `200 OK`.
- Casos malformados o incompletos que deben retornar `400 Bad Request`.
- Rutas inexistentes que deben retornar `404 Not Found`.
- Situaciones forzadas de error interno que deben retornar `500 Internal Server Error`.

Cada request incluye la validación automática del código de estado y contenido esperado. La colección será ejecutada localmente con el servidor corriendo en `localhost:7878`.

D. Pruebas de Carga y Concurrencia

Se utilizará Postman en modo Runner para simular múltiples conexiones concurrentes.

Las pruebas de carga se realizarán sobre:

- Rutas de procesamiento rápido (`/fibonacci`, `/toupper`).
- Rutas que simulan tareas concurrentes (`/simulate`, `/loadtest`).
- Rutas con acceso a sistema de archivos (`/createfile`, `/deletefile`).

Los indicadores observados serán:

- Tiempo promedio de respuesta.
- Número de solicitudes procesadas por segundo.
- Comportamiento bajo saturación del thread pool.

E. Cobertura de Código

Se utilizará `cargo tarpaulin` para medir la cobertura de pruebas unitarias y de integración. El objetivo es alcanzar una cobertura mínima del 90% del código fuente, tal como lo exige la rúbrica del proyecto.

F. Evidencia de Pruebas

Los resultados de las pruebas se documentarán mediante:

- Capturas de pantalla de Postman y consola.
- Registros de ejecución (`.txt` o `.log`).
- Reportes generados por herramientas de cobertura.

V. Resultados Experimentales

En esta sección se presentan los resultados obtenidos al ejecutar las pruebas unitarias, de integración, funcionales y de carga sobre el servidor HTTP concurrente desarrollado. Se emplearon herramientas como Postman y su módulo de colección Runner para automatizar los requests y registrar los tiempos de respuesta, los códigos HTTP retornados y el comportamiento bajo múltiples solicitudes simultáneas.

A. Pruebas Unitarias

Se implementaron pruebas unitarias para las funciones principales del sistema, con el objetivo de verificar que cada módulo individual funcione correctamente de manera aislada. Estas pruebas se desarrollaron dentro del archivo `mod.rs` de cada módulo, utilizando la macro `#[test]`.

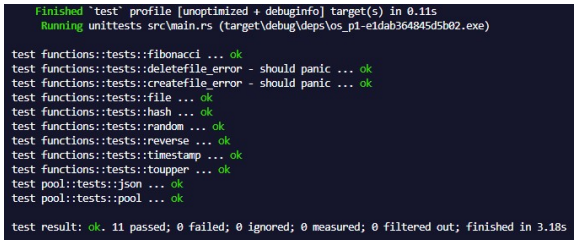

Fig. 1. Resultado de ejecución de pruebas unitarias con `cargo test`

En la Figura 1 se observa que se ejecutaron un total de 11 pruebas, todas ellas pasaron exitosamente sin fallos ni excepciones. Las funciones probadas incluyen operaciones como `fibonacci`, `reverse`, `timestamp`, y la verificación de errores esperados en funciones como `createfile` y `deletefile` (las cuales se validaron con tests que deben generar `panic!()` bajo ciertas condiciones).

Este resultado evidencia que los módulos individuales del sistema funcionan según lo esperado, validando correctamente tanto escenarios normales como de error controlado. Las pruebas unitarias contribuyen a asegurar la estabilidad de las funcionalidades antes de evaluar su integración con otros componentes.

B. Pruebas de integración

Aunque las pruebas de integración no se realizaron como una fase aislada, su verificación se encuentra incorporada de manera implícita dentro de las pruebas funcionales y de carga presentadas. Las ejecuciones mostradas en las Figuras 1 a 6 implican la interacción entre los diferentes componentes del servidor: la recepción y análisis de solicitudes HTTP, el enrutamiento a los módulos funcionales correctos, la ejecución concurrente en hilos, y la construcción final de las respuestas.

Cada solicitud representó un escenario realista que valida el flujo completo del sistema, incluyendo tanto rutas exitosas como errores controlados. En conjunto, estas pruebas confirman que el servidor está correctamente integrado, con una arquitectura capaz de gestionar concurrencia, entradas inválidas, y operaciones sincronizadas entre múltiples rutas.

C. Pruebas funcionales

Las pruebas funcionales tuvieron como objetivo verificar que cada uno de los endpoints del servidor respondiera correctamente tanto ante solicitudes válidas como ante entradas mal formateadas o rutas inexistentes.

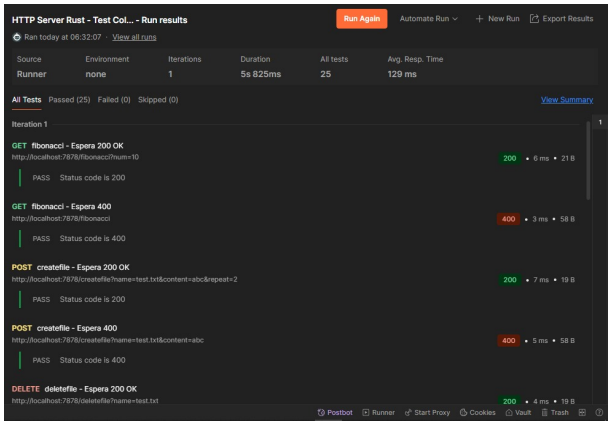

Fig. 2. Pruebas funcionales - Iteración única de 25 requests

En la Figura 2 se muestran 25 solicitudes distribuidas en distintas rutas, incluyendo casos válidos (e.g. `/fibonacci`, `/createfile`) y errores controlados (e.g. `/deletefile` sin archivo existente, o parámetros faltantes). Todos los endpoints respondieron con el código de estado esperado, incluyendo respuestas 200, 400, 404 y 500, según el caso. El tiempo promedio de respuesta fue de 129 ms.

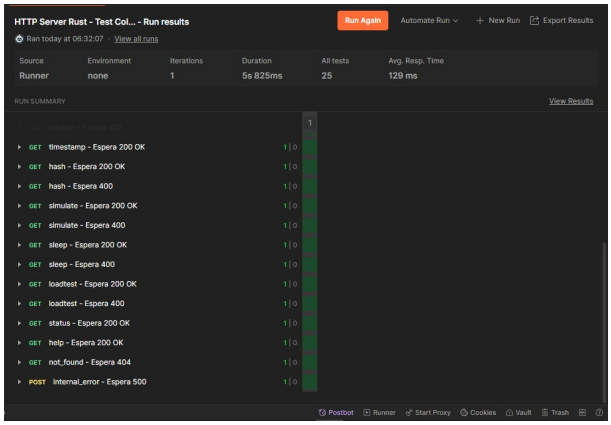

Fig. 3. Resumen de pruebas funcionales - 100% éxito

La Figura 3 presenta el resumen de la ejecución. Todos los tests pasaron exitosamente, sin fallos ni respuestas inesperadas.

D. Pruebas de carga

Las pruebas de carga se realizaron con dos enfoques: uno centrado en un solo endpoint intensivo (`/loadtest`), y otro combinando múltiples rutas de forma concurrente.

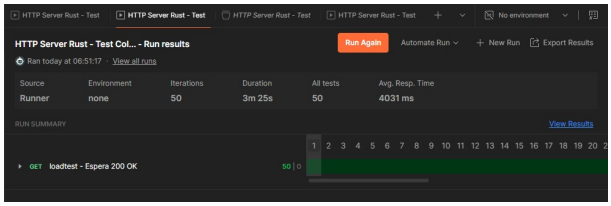

Fig. 4. Prueba de carga - 50 iteraciones a `/loadtest`

La Figura 4 muestra una prueba de carga ejecutando 50 veces el endpoint `/loadtest?tasks=10&sleep=1`. Cada iteración simula 10 tareas de espera concurrentes, lo cual estresa directamente el pool de hilos del servidor. El tiempo promedio de respuesta fue de 4031 ms, consistente con el tiempo estimado por el parámetro `sleep`.

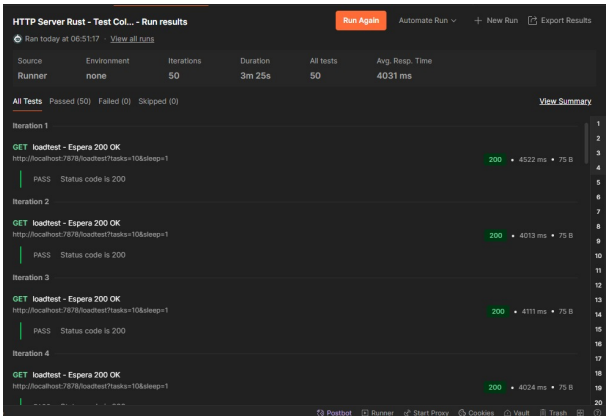

Fig. 5. Detalle de iteraciones de `/loadtest` - 100% éxito

La Figura 5 detalla el resultado individual de varias de las iteraciones de la prueba de carga. Todas las respuestas fueron exitosas (código 200), sin errores ni caídas del servidor. Esto demuestra un comportamiento estable incluso bajo saturación parcial del sistema.

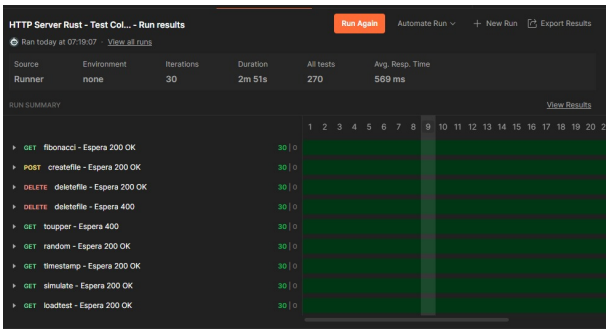

Fig. 6. Prueba de carga variada - 30 iteraciones, múltiples rutas

La Figura 6 muestra una ejecución de 270 requests divididos en 30 iteraciones de una colección compuesta por múltiples endpoints (como `/fibonacci`, `/simulate`, `/createfile`, entre otros). Se simula así una carga más realista y variada. El servidor mantuvo una media de respuesta de 569 ms, sin errores.

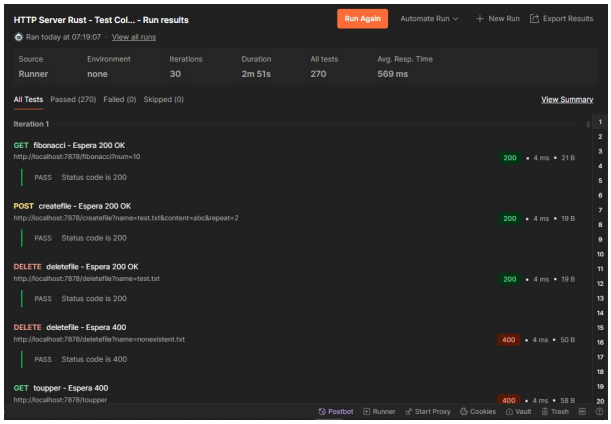

Fig. 7. Detalle de iteraciones variadas - sin fallos

La Figura 7 presenta el detalle de los resultados de la prueba mixta. Se observa que todos los endpoints respondieron correctamente, incluyendo aquellos que simulan errores (como 400 y 500), lo cual indica que el servidor maneja adecuadamente la concurrencia y mantiene la disponibilidad bajo uso intensivo.

VI. Discusión

El desarrollo de este servidor HTTP permitió validar en la práctica múltiples conceptos fundamentales de los sistemas operativos, destacando la correcta implementación de un modelo de concurrencia seguro, el uso eficiente de recursos del sistema y la construcción modular del software. A través de una arquitectura basada en pools de hilos y parsing manual de solicitudes, se logró un control total sobre la ejecución, lo cual es especialmente útil en entornos donde se requiere entender el comportamiento del sistema a bajo nivel.

Entre los aspectos más destacados se encuentran:

- **Robustez del diseño:** El servidor se mantuvo estable bajo diversas cargas, incluyendo pruebas de estrés y uso intensivo de la cola de tareas. Esto demuestra que el modelo de concurrencia implementado es resiliente y evita condiciones de carrera.
- **Modularidad efectiva:** La separación de cada comando en módulos independientes dentro del directorio "functions/" facilitó la implementación, prueba, mantenimiento y potencial extensión futura del sistema.
- **Comprobación exhaustiva mediante Postman:** La colección de pruebas creada en Postman permitió validar de manera sistemática cada ruta y escenario esperado, incluyendo casos de éxito, errores de entrada, rutas inexistentes y errores internos.
- **Limitaciones asumidas:** A pesar de no utilizar programación asíncrona, el modelo de hilos y canales demostró ser suficiente para el alcance del proyecto y se mantuvo dentro de los lineamientos del curso.

Por otro lado, también se identificaron desafíos relevantes:

- El manejo manual del protocolo HTTP implicó una carga significativa en validación, delimitación de parámetros y generación correcta de respuestas.
- La creación de un pool de hilos personalizado, si bien valiosa como aprendizaje, puede ser propensa a errores si no se realiza con cuidado.

En conjunto, los resultados obtenidos indican que el servidor cumple con los objetivos técnicos y pedagógicos, ofreciendo una base sólida para proyectos posteriores en redes, pruebas automatizadas o benchmarking de sistemas distribuidos.

VII. Conclusiones

El proyecto permitió aplicar de forma práctica los conceptos de sistemas operativos, especialmente en áreas como concurrencia, manejo de procesos, comunicación entre hilos y arquitectura cliente-servidor. Se logró implementar un servidor HTTP desde cero utilizando exclusivamente bibliotecas estándar en Rust.

Los aprendizajes adquiridos incluyen:

- Diseño e implementación de un servidor TCP concurrente.
- Manejo seguro de memoria y sincronización de hilos.
- Pruebas funcionales y diseño de rutas HTTP robustas.
- Validación de entradas, control de errores y modularidad del código.

El servidor puede extenderse fácilmente a nuevos comandos, y su estructura modular permite que sea utilizado como base para proyectos más complejos.

References

[1] Steve Klabnik y Carol Nichols, *The Rust Programming Language*, 2019. Disponible en: https://doc.rust-lang.org/book/

[2] Abraham Silberschatz, Peter Baer Galvin y Greg Gagne, *Operating System Concepts*, Wiley, 10ª edición, 2018.

[3] Wikipedia, *List of HTTP status codes*. Disponible en: https://en.wikipedia.org/wiki/List_of_HTTP_status_codes

[4] Rust Documentation, *std::net – Networking primitives*. Disponible en: https://doc.rust-lang.org/std/net/index.html

[5] Rust Documentation, *std::sync – Synchronization primitives*. Disponible en: https://doc.rust-lang.org/std/sync/index.html