Implementación de un Servidor HTTP Concurrente desde Cero en Go

John Sánchez Cespedes 2021080092
Valeria Gómez Acuña 2022173229
Escuela de Ingeniería en Computación
Tecnológico de Costa Rica
Email: jostsace@estudiantec.cr, vale@estudiantec.cr

***Abstract*—Este trabajo presenta el diseño e implementación de un servidor HTTP concurrente utilizando el lenguaje Go y sockets TCP de bajo nivel. Se aborda la motivación técnica del proyecto, los fundamentos operativos aplicados y se discuten resultados experimentales obtenidos mediante pruebas de carga y robustez. Se logró un servidor funcional con ruteo personalizado, manejo de concurrencia, sincronización mediante mutex y arquitectura modular.**

I. Introducción

En el contexto de los cursos de Sistemas Operativos, comprender los mecanismos de comunicación en red, concurrencia y sincronización es fundamental. Este proyecto simula el desarrollo real de una herramienta de infraestructura, aplicando estos conceptos para construir un servidor HTTP que responde a solicitudes GET sin utilizar bibliotecas de alto nivel.

II. Marco Teórico

A. Servidores HTTP

El protocolo HTTP (Hypertext Transfer Protocol) es la base de la comunicación en la World Wide Web. Define cómo los clientes (como navegadores o herramientas como curl) se comunican con los servidores mediante solicitudes y respuestas estructuradas en texto plano. En su versión HTTP/1.0, el protocolo establece una conexión por cada solicitud, utilizando típicamente el verbo `GET` para recuperar recursos desde el servidor.

Un servidor HTTP es una aplicación que escucha en un puerto determinado y se encarga de recibir solicitudes HTTP, interpretarlas, ejecutar la acción asociada y devolver una respuesta con un código de estado (por ejemplo, 200 OK, 404 Not Found). Este servidor puede manejar distintos tipos de recursos como archivos, procesamiento de datos o ejecución de comandos. En este proyecto, se construye un servidor HTTP personalizado desde cero, sin el uso de bibliotecas embebidas como `net/http`, siguiendo fielmente el protocolo HTTP/1.0.

B. Sockets y Comunicación en Red

La comunicación entre procesos en red se fundamenta en el uso de *sockets*, que son estructuras de software que permiten la transmisión de datos entre dos extremos conectados. En el modelo TCP/IP, un socket TCP permite establecer una conexión fiable entre un cliente y un servidor, garantizando el orden y la entrega de los datos transmitidos.

Desde el punto de vista de la implementación en Go, el paquete `net` proporciona una interfaz de bajo nivel para trabajar con sockets. La función `net.Listen` permite que el servidor abra un puerto y escuche conexiones entrantes, mientras que `Accept` establece una conexión con cada cliente. Posteriormente, se utilizan las operaciones `Read` y `Write` sobre la conexión establecida (`net.Conn`) para intercambiar datos.

Este mecanismo es clave en la arquitectura cliente-servidor del proyecto, ya que cada conexión HTTP establecida con el servidor representa un socket activo que debe ser manejado de forma concurrente.

C. Concurrencia y Procesos

La concurrencia es un aspecto esencial en los sistemas operativos modernos y en servidores de red, donde múltiples solicitudes pueden llegar simultáneamente. En lenguajes como C o Java, la concurrencia se logra mediante procesos o hilos (threads). En Go, se introducen las *goroutines*, que son unidades ligeras de ejecución administradas por el propio runtime del lenguaje.

Una goroutine se ejecuta de forma concurrente con otras, permitiendo que el servidor pueda atender múltiples clientes sin necesidad de bloquearse. Esto permite una alta eficiencia y aprovechamiento de los recursos del sistema.

Para manejar el acceso seguro a recursos compartidos (como archivos, contadores de conexiones o estructuras de estado), Go proporciona mecanismos de sincronización como `sync.Mutex`, que permiten implementar secciones críticas. En este proyecto, se utiliza un mutex global para controlar el acceso a la carpeta `files/`, evitando condiciones de carrera en operaciones como la creación y eliminación de archivos concurrentes.

Gracias a estas herramientas, el servidor puede escalar horizontalmente y responder de forma robusta a múltiples solicitudes simultáneas.

D. Arquitectura Cliente-Servidor

La arquitectura cliente-servidor es un modelo fundamental en el desarrollo de sistemas distribuidos, donde se establece una clara división de roles entre los componentes que solicitan servicios (clientes) y aquellos que los proveen (servidores). Este modelo permite la interacción entre múltiples dispositivos

o procesos a través de una red, promoviendo escalabilidad, modularidad y separación de responsabilidades.

En este esquema, el **cliente** inicia la comunicación al enviar una solicitud estructurada (por ejemplo, una petición HTTP), mientras que el **servidor** permanece en escucha, esperando conexiones entrantes. Una vez recibida la solicitud, el servidor la procesa, ejecuta la acción correspondiente y retorna una respuesta.

En el contexto del proyecto desarrollador, el servidor HTTP implementado en Go funciona como el componente central del modelo. Escucha conexiones entrantes a través de sockets TCP y procesa cada solicitud en una goroutine independiente, respondiendo según la ruta solicitada. Por otro lado, el cliente puede ser una herramienta como `curl` o Postman, que se conecta al servidor para consumir los servicios expuestos (como `/fibonacci`, `/reverse`, entre otros).

Este modelo no solo facilita la concurrencia y el aislamiento de componentes, sino que también permite simular situaciones reales donde múltiples usuarios interactúan simultáneamente con una misma aplicación servidor, haciendo énfasis en la importancia de la gestión de recursos, sincronización y control de concurrencia en sistemas operativos.

III. Diseño e Implementación

A. Estructuras Clave

- **Request**: Contiene la conexión, ruta, parámetros y canal de notificación.
- **Server**: Gestiona pools de workers, métricas, el socket TCP y shutdown controlado.
- **Metricas**: Estructura protegida con mutex para contar solicitudes y tiempo de actividad.
- **FastPool**: Es una pool de trabajadores que pertenece al servidor, los cuales realizan tareas que no tardan mucho, los comandos ejecutados por el FastPool son /help, /timestamp, /reverse, /toupper, /hash y /random. La cantidad de trabajadores por cada pool es configurable, por lo que se pueden establecer más o menos de ser necesario.
- **SlowPool**: Es una pool de trabajadores que pertenece al servidor, los cuales realizan tareas que pueden tardar mucho, incluyendo varios segundos, los comandos enviados hacia el SlowPool son /fibonacci, /simulate, /sleep, /loadtest, /createfile, /deletefile. La cantidad de trabajadores es configurables.
- **Worker**: Es el encargado de ejecutar los comandos, cuando recibe una tarea su estado cambia a ”ocupado” y cuando la finaliza su estado vuelve a ”disponible”

B. Módulos Implementados

Cada ruta tiene su handler separado dentro del paquete ‘handlers/‘. Las rutas más relevantes incluyen ‘/fibonacci‘, ‘/reverse‘, ‘/createfile‘, ‘/status‘ y ‘/loadtest. Cada uno de ellos recibe los parametros correspondientes y verifican que los parametros tengan el formato adecuado, por ejemplo que sea un número entero positivo o que el texto no sean comillas vacías. Estos handlers/ se encargan de retornar la respuesta de la solicitud, ya sea el resultado o un mensaje de error con su código correspondiente.

IV. Estrategia de Pruebas

Se utilizó ‘curl‘ para pruebas manuales y ‘go test‘ para pruebas unitarias. Se verificó el correcto funcionamiento de rutas con parámetros válidos e inválidos, manejo de concurrencia y bloqueo con mutex en el acceso a archivos. Las pruebas realizadas se encuentran especificadas en el archivo llamado unitTest.sh, el cual al ejecutarlo realiza todas las solicitudes correspondientes con los diferentes parametros.

V. Resultados Experimentales

El servidor demostró estabilidad bajo carga concurrente (hasta 50 conexiones simultáneas). Las pruebas de latencia con ‘/sleep‘ y de procesamiento con ‘/simulate‘ permitieron observar comportamiento bajo diferentes escenarios. Al realiza solicitudes erroneas el servidor logra manejarlas y retornar un mensaje de error correspondiente al tipo de error.

VI. Discusión

El diseño basado en workers especializados permitió escalabilidad horizontal y control eficiente de la carga. Cada worker actúa como una entidad responsable de procesar solicitudes de un tipo determinado, permitiendo la reutilización de goroutines sin saturar los recursos del sistema.

Durante el desarrollo del sistema, una de las decisiones más desafiantes fue definir cómo manejar la concurrencia de forma robusta. Inicialmente, se consideró lanzar una goroutine nueva por cada solicitud entrante, lo cual funcionaba correctamente en escenarios con pocas conexiones. Sin embargo, esta estrategia escalaba mal ante cargas elevadas: un volumen alto de solicitudes simultáneas podía generar miles de goroutines concurrentes, lo que eventualmente colapsaría el servidor por agotamiento de recursos (memoria, CPU y descriptores de archivo).

La solución adoptada fue la implementación de un modelo de *worker pool*. En este enfoque, se crean una cantidad fija de workers (goroutines) que permanecen en espera de solicitudes. Estas solicitudes se enrutan a través de canales hacia los workers disponibles. De esta forma se mantiene la concurrencia, pero de manera controlada y predecible, eliminando el riesgo de sobrecarga por creación masiva de goroutines.

Adicionalmente, se identificó la necesidad de validar referencias nulas (`nil`) en estructuras como `Request` para evitar errores de tipo `panic` en tiempo de ejecución. Otro aspecto crítico fue la sincronización de estructuras compartidas, como las métricas globales y el acceso a la carpeta `files/`, lo cual se resolvió mediante el uso de `sync.Mutex`.

En conjunto, estas decisiones permitieron que el servidor sea funcional, estable bajo carga y alineado con principios de diseño seguro y eficiente en sistemas operativos.

VII. Conclusiones

El desarrollo de este proyecto representó una experiencia integral en la aplicación de conceptos fundamentales de sistemas operativos, tales como el manejo de sockets, la concurrencia controlada, la sincronización de recursos compartidos y el diseño modular de software.

Se logró implementar un servidor HTTP funcional desde cero utilizando el lenguaje Go, sin dependencias de bibliotecas de alto nivel. Uno de los principales aprendizajes fue que, si bien Go facilita la concurrencia mediante goroutines, su uso sin restricciones puede conducir a saturación del sistema. Por ello, se optó por una arquitectura basada en pools de workers especializados, lo que permitió mantener la concurrencia dentro de límites seguros y predecibles, asegurando la estabilidad del servidor bajo condiciones de carga.

Asimismo, se integraron mecanismos de sincronización con `sync.Mutex` para proteger secciones críticas, como el acceso concurrente a archivos y el registro de métricas globales. También se aplicaron validaciones robustas para evitar errores comunes, como el acceso a referencias nulas y entradas mal formadas.

El resultado final fue un sistema estable, extensible y alineado con buenas prácticas de ingeniería de software. Este proyecto no solo consolidó conocimientos teóricos, sino que también reforzó habilidades prácticas en el desarrollo de soluciones concurrentes, eficientes y seguras.

VIII. Referencias

- Go Documentation: https://golang.org/doc/
- RFC 1945 - HTTP/1.0: https://datatracker.ietf.org/doc/html/rfc1945
- Sistemas Operativos Modernos - Andrew S. Tanenbaum
- Effective Go: https://go.dev/doc/effective_go