Instituto Politécnico Nacional
IPN
Escuela Superior de Cómputo
ESCOM

Aplicaciones para comunicaciones en red
Grupo 6CM2 - Ciclo 2026A

Práctica 4

“Desarrollo de un Servidor HTTP Robusto: Conexiones, Tipos MIME y Balanceo de Carga”

Integrantes:

Alexandra Ortiz Villaseñor
Arely Amairani Cruz Rodríguez

Profesor: Axel Ernesto Moreno Cervantes

Arquitectura y Escalabilidad en la Programación de Servidores HTTP

1. La Interfaz con el Mundo: Implementación de los Métodos HTTP Fundamentales

La práctica comienza con la obligación de manejar los cuatro verbos HTTP principales: `GET`, `POST`, `PUT` y `DELETE`. La correcta implementación de estos métodos es esencial para crear un servidor que actúe como un verdadero *backend* para una aplicación web o una API RESTful.

1.1. Manejo de Peticiones y Mapeo de Recursos

El servidor debe tener un **enrutador (router)** interno que inspeccione la línea de solicitud para determinar el método y la URI (Uniform Resource Identifier).

- `GET` **(Lectura):** Es la petición más común. Debe ser **idempotente** (múltiples llamadas producen el mismo resultado sin efectos secundarios en el servidor). La implementación implica leer un recurso del disco o de una base de datos y retornarlo al cliente, generalmente con el código de estado `200 OK`.
- `POST` **(Creación):** Se utiliza para enviar datos y crear un nuevo recurso. No es idempotente. El servidor debe parsear el cuerpo de la petición y, si la creación es exitosa, responder con el código `201 Created` y, a menudo, la ubicación (`Location` *header*) del nuevo recurso.
- `PUT` **(Actualización/Reemplazo):** Se utiliza para reemplazar la totalidad de un recurso existente con el contenido del cuerpo de la petición. Es idempotente. La respuesta típica es `200 OK` si se actualiza, o `201 Created` si el recurso no existía y se crea.
- `DELETE` **(Eliminación):** Se utiliza para eliminar un recurso específico. Es idempotente. La respuesta habitual es `204 No Content` (si la acción fue ejecutada y no hay cuerpo para retornar) o `200 OK`.

1.2. La Importancia Crítica de los Tipos MIME

El requisito de manejar al menos cuatro tipos MIME distintos (`text/html`, `application/json`, `image/jpeg`, etc.) impone una capa de lógica para la serialización y deserialización de datos. El servidor debe ser consciente del encabezado `Content-Type` para peticiones entrantes y del encabezado `Accept` para determinar el formato de la respuesta.

Tipo MIME	Uso Típico	Requisito del Servidor
`text/html`	Páginas web	Debe poder leer archivos `.html` y enviarlos.
`application/json`	APIs RESTful	Necesita una biblioteca o función para serializar/deserializar objetos a formato JSON.

Tipo MIME	Uso Típico	Requisito del Servidor
`image/jpeg`	Contenido binario	Debe leer el contenido del archivo de imagen en bytes y enviarlo sin modificar.
`application/xml`	Intercambio de datos	Requiere un *parser* XML para procesar la estructura de datos.

El desafío reside en programar la lógica que, por ejemplo, al recibir un `POST`, sepa si el cuerpo debe ser tratado como JSON, XML o datos de formulario.

---

2. Fundamentos de Concurrencia: El Pool de Conexiones (`Thread Pool`)

El *pool* de conexiones es la técnica de concurrencia elegida para maximizar el rendimiento y la estabilidad. En lugar de crear un *thread* (hilo de ejecución) nuevo para cada petición (lo cual es lento y consume mucha memoria), el servidor mantiene un conjunto fijo de *threads* listos para trabajar.

2.1. Estructura y Funcionamiento del Pool

El *pool* consta de dos componentes principales:

1. **Cola de Tareas (*Task Queue*):** Una estructura de datos (típicamente una cola concurrente *FIFO*) donde las peticiones entrantes se colocan a la espera de ser procesadas.
2. **Conjunto de Trabajadores (*Worker Threads*):** Un número fijo (*N*) de *threads* que extraen tareas de la cola y las ejecutan.

El tamaño del *pool* (*N*, definido por el usuario) es crucial. Si *N* es muy pequeño, la cola se llena rápidamente; si es muy grande, el *overhead* de la conmutación de contexto del sistema operativo ralentiza el rendimiento (la "contención de recursos").

2.2. Mecanismos de Sincronización

La implementación del *pool* en un lenguaje como Java, Python o C++ requiere el uso de primitivas de sincronización para garantizar la seguridad:

- **Mutexes/Locks:** Para proteger la cola de tareas y evitar que dos *workers* intenten acceder o modificarla al mismo tiempo.
- **Variables de Condición (*Condition Variables*):** Para permitir que los *workers* que no tienen trabajo se "duerman" y sean "despertados" solo cuando se añade una nueva tarea a la cola, optimizando el uso de la CPU.

---

3. Escalabilidad Horizontal: El Mecanismo de Desborde y Redireccionamiento

Esta es la funcionalidad más avanzada de la práctica, simulando un sistema de **balanceo de carga basado en umbrales de uso**.

3.1. Detección de Umbral y Activación

El servidor primario debe monitorear continuamente su carga, definida como el número de *threads* actualmente activos en el *pool*.

- **Capacidad Máxima (N):** Tamaño total del *pool*.
- **Umbral de Activación:** $N/2$ (la mitad de la capacidad).

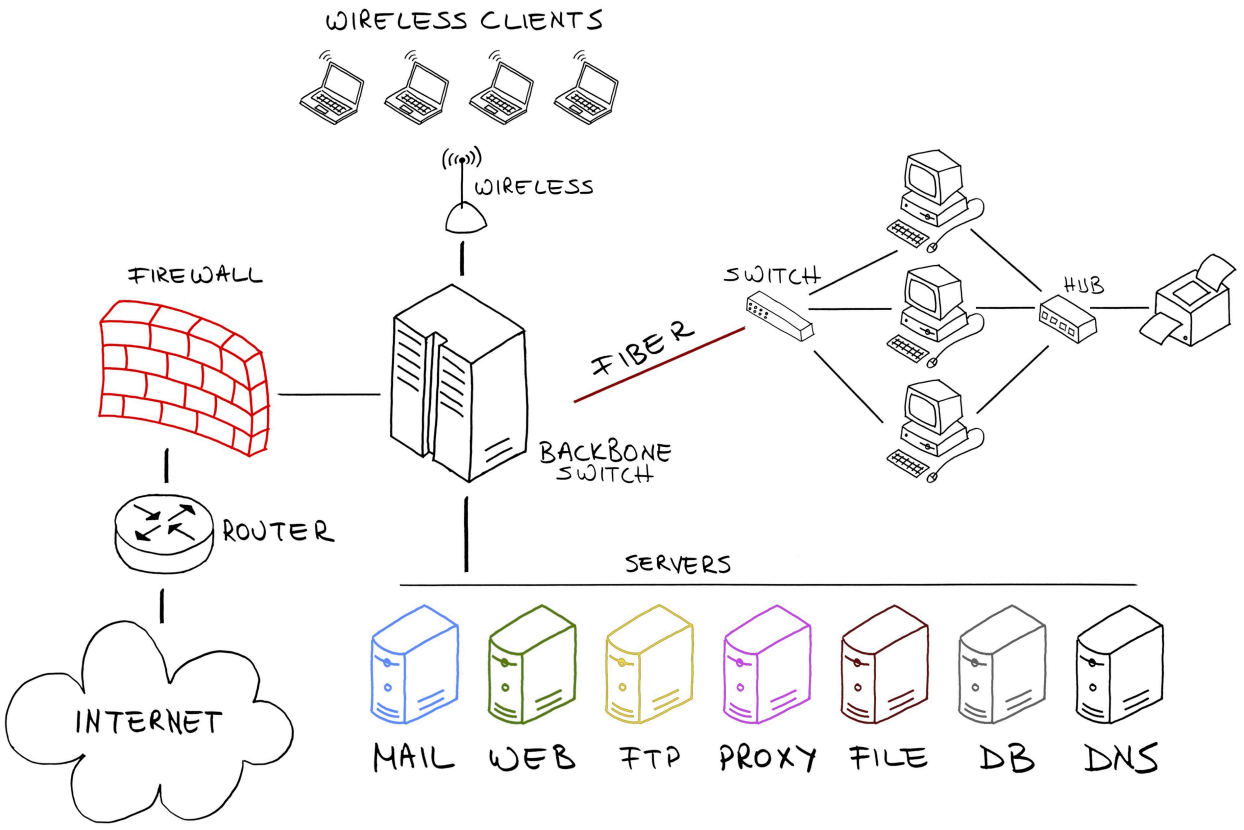
Cuando el número de peticiones activas supera $N/2$, el servidor debe ejecutar el siguiente protocolo:

1. **Verificación de Estado:** Comprobar si el servidor secundario ya está en funcionamiento.
2. **Lanzamiento:** Si el secundario no está activo, el primario debe iniciarlo, asignándole un puerto diferente (ej. Primario en puerto 8080, Secundario en puerto 8081). Esto podría requerir una llamada a un proceso o *thread* separado.

3.2. Redireccionamiento y Proxy

Una vez que el secundario está activo y el umbral se mantiene por encima de $N/2$, las nuevas peticiones deben ser manejadas por el segundo servidor. Existen dos maneras principales de lograr esto:

1. **Redirección HTTP (Código 307/308):** El servidor primario responde al cliente con un código de estado de redirección (`307 Temporary Redirect` o `308 Permanent Redirect`), indicando la nueva URL o puerto (ej. `http://servidor:8081/recurso`). El **cliente es el responsable** de hacer la segunda petición al nuevo servidor. Esto añade latencia.
2. **Proxy Inverso (*Forwarding*):** El servidor primario actúa como un **proxy inverso**. Recibe la petición, la reenvía internamente al servidor secundario, espera la respuesta del secundario y luego la reenvía de vuelta al cliente original. Este método es transparente para el cliente y es el más común en soluciones de balanceo de carga real.



La práctica, al ser de bajo nivel, probablemente se enfoque en la implementación del **Proxy Inverso** o, más simplemente, en el *forwarding* interno de la *socket* de la petición al nuevo puerto. Este mecanismo garantiza que el sistema pueda manejar picos de tráfico sin comprometer la velocidad de procesamiento de las peticiones ya en curso en el *pool* primario.

Conclusión

La programación de este servidor HTTP va más allá de un ejercicio de *networking* básico. Es una síntesis de **protocolo** (HTTP y MIME), **concurrencia** (Thread Pools) y **arquitectura de escalabilidad** (Balanceo de Carga/Redireccionamiento). El desarrollador debe equilibrar la eficiencia del procesamiento de datos (MIME y verbos HTTP) con la estabilidad del sistema (gestión del *pool*). Al lograr que el sistema lance y utilice dinámicamente un servidor secundario, se demuestra una comprensión fundamental de cómo los sistemas modernos crecen para satisfacer una demanda variable y evitar el punto único de falla. El resultado es un *software* robusto, concurrente y diseñado para la escalabilidad.

Referencias

- **Fielding, R., Nottingham, M., & Reschke, J.** (2022). *HTTP Semantics (RFC 9110)*. IETF.
- **Silberschatz, A., Galvin, P. B., & Gagne, G.** (2018). *Operating System Concepts* (10th ed.). Wiley.

- **Tanenbaum, A. S., & van Steen, M.** (2017). *Distributed Systems* (3rd ed.). CreateSpace.