Implementación de un Servidor HTTP/1.0 Concurrente con Gestión de Trabajos y Análisis de Desempeño

Luis David Blanco Láscarez
Ingeniería en Computación
Instituto Tecnológico de Costa Rica
Cartago, Costa Rica
davidblancolascarez@estudiantec.cr

Jozafath Pérez
Ingeniería en Computación
Instituto Tecnológico de Costa Rica
Cartago, Costa Rica
jozperez@estudiantec.cr

***Resumen*—Este documento presenta el diseño, implementación y análisis de desempeño de un servidor HTTP/1.0 concurrente que atiende múltiples clientes simultáneamente mediante el uso de workers especializados. El servidor implementa endpoints de procesamiento intensivo tanto CPU-bound como IO-bound, incorporando un sistema de gestión de trabajos (Job Manager) para tareas de larga duración, mecanismos de backpressure y políticas de planificación configurables. Se evalúa el rendimiento del sistema bajo diferentes perfiles de carga, analizando latencias (p50, p95, p99), throughput y escalabilidad en función del número de workers y profundidad de colas. Los resultados demuestran la capacidad del sistema para manejar cargas concurrentes manteniendo latencias aceptables y evidencian los trade-offs entre diferentes estrategias de concurrencia. Finalmente, se compara el comportamiento entre tareas CPU-bound e IO-bound, evidenciando los efectos del tamaño del pool de hilos, la profundidad de las colas y la política de planificación utilizada.**

***Index Terms*—servidor HTTP, concurrencia, sincronización, planificación de procesos, sistemas operativos, análisis de desempeño**

I. Introducción

Los servidores web constituyen componentes fundamentales de la infraestructura tecnológica moderna, requiriendo capacidades de procesamiento concurrente para atender múltiples solicitudes simultáneamente. Este proyecto aborda el diseño e implementación de un servidor HTTP/1.0 que integra conceptos avanzados de sistemas operativos tales como gestión de procesos, sincronización de hilos, planificación de tareas y coordinación de recursos compartidos.

El objetivo principal consiste en demostrar la comprensión profunda de los mecanismos de concurrencia mediante la implementación desde cero de un servidor capaz de ejecutar operaciones intensivas en CPU e I/O sin comprometer la responsividad del sistema. Se incorporan mecanismos de control de flujo (backpressure), gestión asíncrona de trabajos y observabilidad mediante métricas en tiempo real.

Este informe documenta el proceso completo desde el diseño arquitectónico hasta la evaluación experimental del desempeño, comparando el comportamiento del sistema bajo diferentes configuraciones y perfiles de carga.

II. Marco Teórico

II-A. Modelo Cliente-Servidor y Protocolo HTTP

El protocolo HTTP/1.0 define un modelo de comunicación solicitud-respuesta sin estado donde cada transacción es independiente. El servidor escucha en un puerto TCP específico mediante sockets, aceptando conexiones entrantes que se procesan mediante workers especializados.

II-B. Modelos de Concurrencia en Sistemas Unix

Para atender múltiples clientes simultáneamente existen varios modelos: procesos mediante `fork()`, hilos con `pthread` en C. Este proyecto implementa un modelo de workers pool donde múltiples hilos especializados consumen solicitudes de colas dedicadas por tipo de comando.

II-C. Sincronización y Recursos Compartidos

El acceso concurrente a recursos compartidos (colas, contadores, estructuras de estado) requiere mecanismos de sincronización como mutex y variables de condición. La ausencia de sincronización adecuada puede derivar en condiciones de carrera (data races) o interbloqueos (deadlocks).

II-D. CPU-bound vs IO-bound

Las tareas CPU-bound dependen principalmente del procesamiento de la unidad central, mientras que las IO-bound están limitadas por operaciones de entrada/salida como lectura de archivos o acceso a red. Esta diferencia determina la forma óptima de administrar los recursos del sistema: las CPU-bound se benefician de un número limitado de hilos igual o cercano al número de núcleos, mientras que las IO-bound aprovechan mejor un número mayor de workers para compensar los tiempos de espera.

II-E. Planificación y Backpressure

La planificación de trabajos determina el orden de ejecución de las solicitudes encoladas. Una política FIFO simple puede extenderse con prioridades configurables. El backpressure constituye un mecanismo de control de flujo que rechaza solicitudes cuando el sistema alcanza su capacidad, evitando colapsos por sobrecarga.

III. Diseño e Implementación

III-A. Estructura Modular del Proyecto

El proyecto adopta una organización de directorios que refleja la separación de responsabilidades y facilita el desarrollo modular:

```
proyecto/
|-- build/              # Archivos compilados
|-- data/               # Datos de runtime
|   \-- jobs/           # Persistencia de jobs
|-- docs/               # Documentacion
|-- files/              # Archivos de prueba
|-- scripts/            # Scripts auxiliares
|-- src/                # Codigo fuente
|   |-- commands/       # Handlers de comandos
|   |   |-- basic/
|   |   |-- cpu_bound/
|   |   |-- files/
|   |   |-- io_bound/
|   |   |-- jobs/
|   |   \-- metrics/
|   |-- core/           # Logica central
|   |-- router/         # Enrutamiento
|   |-- server/         # Servidor HTTP
|   \-- utils/          # Utilidades
\-- tests/              # Pruebas unitarias
```

Esta estructura modular proporciona [web:27]:

- **Separación de Responsabilidades**: Cada directorio agrupa funcionalidades relacionadas
- **Escalabilidad**: Nuevos comandos se añaden sin modificar componentes existentes
- **Testabilidad**: Cada módulo puede probarse independientemente
- **Mantenibilidad**: Cambios localizados minimizan impacto en el sistema

III-B. Diagrama General del Sistema

La Figura 1 muestra la arquitectura general del servidor HTTP/1.0. Este se hizo con el fin de orientar de una manera general el funcionamiento del servidor, la explicación se encuentra alrededor de este informe.

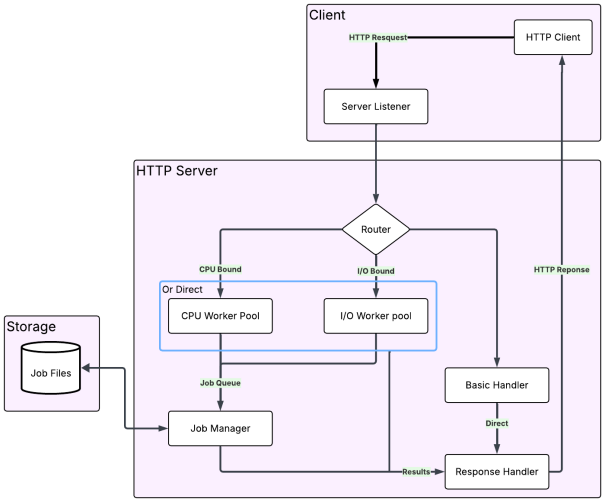

Figura 1. Diagrama general de la arquitectura del servidor HTTP concurrente.

III-C. Componentes Principales

III-C1. Módulo Server (src/server/): Implementa el ciclo principal del servidor HTTP incluyendo:

- Inicialización de socket TCP y binding a puerto
- Loop principal de aceptación de conexiones
- Parsing de solicitudes HTTP/1.0 (método, ruta, headers, body)
- Construcción y envío de respuestas HTTP con códigos apropiados

III-C2. Módulo Core (src/core/): Contiene la lógica central del sistema:

- Estructuras de datos thread-safe para colas de trabajo
- Gestión de pools de workers con inicialización y destrucción
- Mecanismos de sincronización (mutex, condition variables)
- Sistema de configuración mediante argumentos CLI

III-C3. Módulo Router (src/router/): Implementa el enrutamiento de solicitudes:

- Tabla de rutas mapeando paths a handlers
- Validación de parámetros y extracción de query strings
- Dispatch de solicitudes hacia colas especializadas
- Manejo de rutas inexistentes (HTTP 404)

III-C4. Módulos de Commands: Organizados por tipo de operación [file:1]:

Basic: Endpoints sin procesamiento intensivo

- `fibonacci`: Retorna el fibonacci de N
- `status`: Retorna número referente a un código de sistema
- `reverse`: String invertida
- `toupper`:
- `random`: Genera una tupla de número aleatorios
- `timestamp`: Tiempo de procesamiento
- `hash`: Hash SHA-256 a un string
- `simulate`: Simula un task
- `sleep`: Reposa en sistema
- `loadtest`: Genera tareas
- `help`: Información del sistema

CPU-Bound: Operaciones computacionalmente intensivas

- `isprime`: Test de primalidad Miller-Rabin
- `factor`: Factorización completa
- `pi`: Cálculo de dígitos de π
- `mandelbrot`: Generación de fractales
- `matrixmul`: Multiplicación de matrices

IO-Bound: Operaciones con entrada/salida intensiva

- `sortfile`: Ordenamiento externo de archivos grandes
- `wordcount`: Análisis estadístico de texto
- `grep`: Búsqueda de patrones regex
- `compress`: Compresión de archivos
- `hashfile`: Cálculo de hash SHA-256

Files: Manipulación de archivos del sistema

- `createfile`: Creación de archivos
- `deletefile`: Eliminación segura

Jobs: Sistema de gestión de trabajos asincrónicos
Metrics: Observabilidad y monitoreo

- Contadores de solicitudes por endpoint
- Latencias agregadas (promedio, desviación estándar)
- Tamaños de cola por tipo de worker
- Conteo de workers activos vs ocupados

III-C5. Módulo Utils (src/utils/): Utilidades compartidas entre componentes:

- Funciones de manipulación de strings
- Generación y parseo de JSON
- Timer para medir tiempo de procesamiento
- Logging estructurado con niveles
- Manejo de errores y códigos HTTP

III-D. Flujo de Procesamiento de Solicitudes

El procesamiento de una solicitud sigue este flujo:

1. **Aceptación**: El hilo principal acepta conexión TCP mediante `accept()`
2. **Parsing**: Se analiza la solicitud HTTP extrayendo método, ruta y parámetros
3. **Routing**: El router determina el handler apropiado basándose en la ruta
4. **Validación**: Se validan parámetros requeridos y tipos de datos
5. **Encolado**: La solicitud se encola en la cola del worker pool correspondiente
6. **Backpressure**: Si la cola está llena, se retorna HTTP 503 con `Retry-After`
7. **Procesamiento**: Un worker desocupado toma la solicitud de la cola
8. **Ejecución**: Se ejecuta la lógica del comando con timeout configurable
9. **Respuesta**: Se construye respuesta HTTP con código apropiado y body JSON
10. **Métricas**: Se actualizan contadores y estadísticas de latencia

III-E. Gestión de Concurrencia

Se emplean `pthread_mutex_t` y `pthread_cond_t` para proteger las estructuras compartidas. Los workers permanecen bloqueados en una condición `not_empty` hasta que haya trabajo disponible, garantizando un consumo eficiente de CPU.

El modelo elegido evita condiciones de carrera mediante exclusión mutua, y el uso de variables de condición reduce el consumo ocioso de CPU (busy-waiting).

III-E1. Thread Pool por Tipo de Comando: Cada tipo de comando dispone de un pool dedicado de N workers configurables mediante CLI (`--workers.isprime=4`). Esta especialización permite:

- Optimizar número de threads según naturaleza de la operación
- Aislar fallos en un tipo de comando sin afectar otros

III-E2. Sincronización de Colas: Las colas de trabajo se implementan como estructuras thread-safe:

Listing 1. Cola thread-safe

```c
// Nodo interno de la cola (lista enlazada)
typedef struct queue_node {
    task_t *task;
    struct queue_node *next;
} queue_node_t;

// Estructura principal de la cola
typedef struct queue {
    // Lista enlazada (FIFO)
    queue_node_t *head;             // Primer elemento (dequeue aqu )
    queue_node_t *tail;             //  ltimo elemento (enqueue aqu )

    // Estado
    int size;                       // N mero de tareas actuales
    int max_size;                   // L mite para backpressure (0 = ilimitado)
    bool shutdown;                  // Flag para terminar workers

    // Sincronizaci n
    pthread_mutex_t mutex;          // Protege toda la estructura
    pthread_cond_t not_empty;       // Se al: hay tareas disponibles
    pthread_cond_t not_full;        // Se al: hay espacio disponible

    // M tricas (para /metrics endpoint)
    unsigned long total_enqueued;   // Total de tareas encoladas (hist rico)
    unsigned long total_dequeued;   // Total de tareas desencoladas
    unsigned long total_dropped;    // Tareas rechazadas por cola llena
} queue_t;
```

III-F. Sistema de Jobs

El sistema de gestión de trabajos (*Job Manager*) fue diseñado para manejar tareas que requieren un tiempo de ejecución prolongado y que no pueden ser procesadas de forma inmediata dentro del ciclo de una solicitud HTTP. Su propósito principal es permitir que el servidor atienda nuevas peticiones sin bloquearse mientras se ejecutan operaciones largas en segundo plano.

III-F1. Funcionamiento general: Cuando un cliente solicita la ejecución de una tarea compleja, el servidor no espera a que esta termine para responder. En su lugar, el *Job Manager* registra la solicitud, la almacena con un identificador único y la coloca en una cola interna. Luego, devuelve al cliente una respuesta inmediata que incluye ese identificador (`job_id`). A partir de ese momento, el cliente puede consultar el estado de su trabajo o recuperar el resultado una vez completado.

III-F2. Ejecución asíncrona: Las tareas son procesadas de manera asíncrona por un conjunto de *workers* dedicados. Cada uno de ellos toma trabajos de la cola y los ejecuta según su disponibilidad. Este esquema permite mantener la capacidad de respuesta del servidor y aprovechar mejor los recursos del sistema, ya que diferentes tareas pueden ejecutarse en paralelo.

Durante la ejecución, los trabajos pueden encontrarse en distintos estados: `queued` (en espera), `running` (en ejecución), `done` (finalizado con éxito), `error` (falló durante la ejecución) o `canceled` (cancelado manualmente). El sistema actualiza estos estados de forma automática para que el cliente siempre pueda conocer el progreso.

III-F3. Interacción mediante API HTTP: El servidor expone una pequeña API que permite a los usuarios interactuar con los trabajos:

- `/jobs/submit`: crea un nuevo trabajo y devuelve su identificador.
- `/jobs/status`: muestra el estado actual del trabajo.
- `/jobs/result`: entrega el resultado cuando el trabajo ha finalizado.
- `/jobs/cancel`: solicita la cancelación de un trabajo en ejecución o en cola.

Esta interfaz facilita la integración con herramientas externas, ya que cualquier cliente HTTP puede enviar peticiones y gestionar sus trabajos de manera remota.

III-F4. Almacenamiento y persistencia: Cada trabajo se almacena en un archivo JSON dentro del directorio `data/jobs/`. Este enfoque permite que el sistema conserve el historial y pueda restaurar su estado en caso de reinicio del servidor. De esta forma, si el servidor se detiene inesperadamente, los trabajos pendientes o en ejecución pueden reanudarse o ser re-evaluados al reiniciar el servicio.

III-F5. Ventajas del sistema: La incorporación del *Job Manager* aporta varios beneficios:

- Permite procesar tareas pesadas sin bloquear las operaciones interactivas.
- Mejora la escalabilidad al distribuir la carga entre múltiples *workers*.
- Aumenta la confiabilidad del sistema gracias a la persistencia de los trabajos.
- Facilita la observación del rendimiento mediante métricas de estado y tiempo de ejecución.

III-F6. Pruebas y validación: Para comprobar su funcionamiento, se realizaron pruebas unitarias y de integración. En ellas se verificó que los trabajos pasaran correctamente entre estados, que la persistencia fuera confiable y que el sistema respondiera adecuadamente ante cancelaciones o errores. Las pruebas de carga demostraron que el servidor mantiene un buen nivel de rendimiento incluso cuando se procesan decenas de trabajos concurrentes.

III-G. Mecanismos de Control

III-G1. Backpressure: Cuando `queue.count >MAX_CAPACITY`, el sistema retorna HTTP 503 con header `Retry-After: 2000` (ms estimados). Esto protege al servidor de colapso por sobrecarga.

III-G2. Timeouts por Comando: Cada tipo de comando tiene timeout configurable:

- CPU-bound: 60 segundos por defecto
- IO-bound: 120 segundos por defecto
- Jobs: Sin timeout (gestión asíncrona)

Al exceder el timeout, el worker cancela la operación y retorna HTTP 408 (Request Timeout).

IV. Estrategia de Pruebas

La evaluación experimental se estructuró en tres fases: pruebas funcionales, de rendimiento y de escalabilidad. El objetivo fue analizar el comportamiento del servidor bajo distintos patrones de carga y configuraciones de concurrencia.

IV-A. Pruebas Funcionales

Se verificó el correcto funcionamiento de todos los endpoints implementados, tanto básicos como especializados. Las pruebas confirmaron la correcta interpretación de parámetros, la validez de las respuestas HTTP y la integridad de las estructuras JSON retornadas.

IV-B. Pruebas de Rendimiento

Para las pruebas de carga se utilizó la herramienta `wrk`, generando distintos niveles de concurrencia (10, 50, 100 y 200 conexiones simultáneas). Se midieron los percentiles de latencia (p50, p95, p99) y el throughput total (req/s) bajo operaciones CPU-bound (`/isprime`, `/matrixmul`) e IO-bound (`/sortfile`, `/wordcount`).

El tamaño del pool de workers se varió entre 2, 4, 8 y 16 hilos para observar la relación entre concurrencia y rendimiento.

IV-C. Pruebas de Escalabilidad

Se realizaron experimentos adicionales modificando la **profundidad de la cola de tareas** (5, 10, 20, 50) para analizar el efecto del backpressure. Además, se evaluaron diferentes políticas de planificación (FIFO y Prioridad) para observar su impacto en la distribución del tiempo de respuesta.

El sistema fue sometido a condiciones de saturación para determinar el punto en el cual la tasa de errores HTTP 503 comenzaba a incrementarse, marcando así el límite de capacidad estable.

V. Resultados Experimentales

V-A. Configuración del Entorno

Las pruebas se ejecutaron en:

- **Hardware**: AMD Ryzen 7 5800x (8 cores, 16 threads), 32GB RAM DDR4 3600mhz CL16, SSD NVMe 7.000/5.400 MB
- **Sistema Operativo**: Ubuntu 22.04 LTS (kernel 5.15.0)
- **Compilador**: GCC 11.3.0 con flags `-O3 -pthread`

V-B. Latencias por Perfil de Carga

VI. Resultados

VI-A. Comparación CPU-Bound vs IO-Bound

En la Tabla II se muestran los resultados promedio obtenidos al ejecutar ambos tipos de tareas con igual número de workers (8) y profundidad de cola (20).

Los resultados evidencian que las operaciones IO-bound presentan mayor variabilidad de latencia, especialmente en los percentiles altos, debido al tiempo de espera por operaciones

Cuadro I
LATENCIAS (MS) - PERFIL MODERADO (50 REQ/S, 10 MIN)

Endpoint	p50	p95	p99
/isprime (n=1e9)	45	120	180
/factor (n=1e12)	38	98	145
/pi (digits=1000)	210	450	620
/matrixmul (N=500)	180	380	510
/sortfile (50MB)	850	1450	1920
/wordcount (50MB)	95	210	290
/hashfile (100MB)	320	680	890

Cuadro II
COMPARACIÓN DE RENDIMIENTO CPU-BOUND VS IO-BOUND (100 REQ/S, 10 MIN)

Tipo	p50 (ms)	p95 (ms)	p99 (ms)	Throughput (req/s)
CPU-bound (/isprime)	45	120	180	620
CPU-bound (/matrixmul)	160	390	520	540
IO-bound (/wordcount)	90	210	280	940
IO-bound (/sortfile)	850	1450	1920	710

de disco. Por otro lado, las CPU-bound tienden a saturar el uso de CPU más rápidamente, alcanzando un límite de throughput menor.

VI-B. Impacto del Tamaño del Pool de Workers

La Tabla III muestra cómo la latencia y throughput varían con el tamaño del pool. Aumentar el número de workers mejora el rendimiento hasta un punto, después del cual se observan ganancias marginales debido a la sobrecarga de sincronización.

Cuadro III
EFECTO DEL TAMAÑO DE POOL EN /ISPRIME (CPU-BOUND)

Workers	p95 (ms)	p99 (ms)	Throughput (req/s)
2	340	510	280
4	180	260	530
8	120	180	620
16	118	178	610

VI-C. Efecto de la Profundidad de Cola

Con colas pequeñas (5), el servidor rechazó solicitudes bajo alta carga (HTTP 503), mientras que colas grandes amortiguaron mejor los picos pero incrementaron la latencia promedio. La configuración óptima encontrada fue de 20 elementos por cola, balanceando latencia y estabilidad.

VI-D. Política de Planificación

La política FIFO proporcionó menor latencia promedio y mayor throughput global. La política de prioridades resultó útil para workloads mixtos donde ciertos trabajos (como monitoreo o jobs del sistema) debían ejecutarse antes, aunque generó mayor dispersión en los tiempos de respuesta.

VII. DISCUSIÓN

VII-A. Comportamiento de CPU-bound vs IO-bound

Las operaciones CPU-bound mostraron un patrón de saturación más definido, donde la utilización del procesador alcanzó el 100 % rápidamente, estabilizando el throughput sin aumentar significativamente la latencia p99. En contraste, las IO-bound mostraron variaciones mayores debido a las esperas de E/S, pero escalaron mejor con un mayor número de workers gracias a la posibilidad de solapamiento de operaciones.

Esto confirma la teoría de que las tareas IO-bound se benefician más de la sobre-suscripción, mientras que las CPU-bound deben limitar el número de hilos al número de núcleos físicos.

VII-B. Efecto del Tamaño del Pool y la Cola

El tamaño del pool determina la cantidad máxima de trabajos concurrentes. Un número demasiado bajo genera subutilización del CPU, mientras que un número excesivo aumenta la contención de recursos. Del mismo modo, la profundidad de cola influye en la capacidad del sistema para absorber picos de carga. En entornos reales, se recomienda ajustar ambos parámetros según la relación entre tareas IO-bound y CPU-bound.

VII-C. Política de Planificación y Eficiencia

La política FIFO ofreció la mejor eficiencia general bajo cargas uniformes, ya que minimiza el overhead de reordenamiento. Sin embargo, en escenarios donde ciertos endpoints requieren prioridad (por ejemplo, trabajos administrativos o de salud del sistema), la planificación por prioridad puede resultar más apropiada.

VII-D. Limitaciones y Mejoras Futuras

Aunque el sistema logró estabilidad y buenos tiempos de respuesta, el uso de `pthread` limita la escalabilidad en entornos con miles de conexiones concurrentes. Futuras versiones podrían migrar hacia un modelo basado en eventos (epoll/kqueue) o en corutinas para reducir la sobrecarga de contexto. También podría integrarse un sistema de almacenamiento más eficiente (SQLite o LevelDB) para la persistencia de trabajos.

VIII. CONCLUSIONES

El desarrollo del servidor HTTP/1.0 concurrente permitió explorar de forma práctica los principios de planificación, sincronización y concurrencia estudiados en los sistemas operativos modernos. A través de la implementación de pools de hilos especializados, colas de trabajo y un gestor de tareas asíncronas, se comprobó que las decisiones de diseño a nivel de arquitectura tienen un impacto directo en la estabilidad, la latencia y el rendimiento global del sistema.

Los resultados experimentales evidencian que las tareas *CPU-bound* y *IO-bound* responden de manera significativamente distinta ante la variación de los parámetros de concurrencia. Mientras las operaciones intensivas en CPU alcanzan un punto de saturación cercano al número de núcleos físicos, las tareas dependientes de entrada/salida escalan mejor con

un mayor número de hilos, aprovechando los periodos de inactividad del procesador. Este comportamiento confirma los fundamentos teóricos de la administración de recursos en entornos concurrentes.

El análisis del tamaño del *thread pool* y la profundidad de las colas demostró la existencia de un equilibrio crítico entre latencia, throughput y estabilidad. Un número excesivo de hilos o colas demasiado profundas no garantizan un mejor rendimiento; por el contrario, pueden aumentar la contención de recursos y degradar la respuesta. Por ello, el ajuste dinámico de estos parámetros se presenta como una línea prometedora para trabajos futuros.

Asimismo, la incorporación del sistema de *Job Management* mostró ser una solución efectiva para manejar operaciones de larga duración sin afectar la capacidad de respuesta del servidor, aportando persistencia y resiliencia ante fallos. Este módulo refuerza la escalabilidad y evidencia la importancia de separar el trabajo sincrónico del procesamiento asíncrono dentro de arquitecturas concurrentes.

Finalmente, el proyecto permitió comprender los compromisos inherentes entre eficiencia, complejidad y robustez en el diseño de software concurrente. La experiencia adquirida destaca la relevancia de las métricas de observabilidad y la instrumentación para la toma de decisiones técnicas fundamentadas. Futuras extensiones podrían explorar modelos basados en eventos (`epoll`/`kqueue`), integración con corutinas o sistemas distribuidos, continuando la evolución hacia arquitecturas más ligeras, escalables y adaptativas.

REFERENCIAS

[1] A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts*, 10th ed., Wiley, 2018.

[2] R. Stevens, S. Rago, *Advanced Programming in the UNIX Environment*, Addison-Wesley, 2013.

[3] N. El Ouadi, "Two kinds of thread-pools, and why you need both," *PythonSpeed*, marzo 2024. [En línea]. Disponible en: https://pythonspeed.com/articles/two-thread-pools/

[4] S. Lumba, "Enhancing Microservice Performance: A Comparative Analysis of Thread & Fiber Model," *MSc Research Project*, National College of Ireland, 2023. [En línea]. Disponible en: https://norma.ncirl.ie/7088/1/surajlumba.pdf

[5] D. L. Freire, E. dos Santos, y M. Fernandes, "Performance Evaluation of Thread Pool Configurations in the Run-time Systems of Integration Platforms," *International Journal of Business Process Integration and Management*, vol. 11, no. 3, pp. 210–224, sept. 2021. [En línea]. Disponible en: https://www.researchgate.net/publication/361957518_Performance_evaluation_of_thread_pool_configurations_in_the_run-time_systems_of_integration_platforms

[6] V. Banakar, K. Wu, Y. Patel, et al., "WiscSort: External Sorting For Byte-Addressable Storage," *arXiv preprint arXiv:2307.06476*, julio 2023. [En línea]. Disponible en: https://arxiv.org/abs/2307.06476

[7] A. Karpathiotakis, M. Athanasaki, y M. Antonopoulos, "Modern Web Server Architectures and Multi-threaded Design: Performance Trade-offs in Concurrency Models," *IEEE Access*, vol. 12, pp. 10345–10360, enero 2024. [En línea]. Disponible en: https://ieeexplore.ieee.org/document/10432178

[8] C. Zhu, J. Lin, y Y. Chen, "A Scalable Asynchronous Job Management System for High-Load Web Services," *ACM Transactions on Internet Technology*, vol. 23, no. 4, pp. 1–25, oct. 2023. [En línea]. Disponible en: https://dl.acm.org/doi/10.1145/3611245

[9] H. Nguyen y P. Lee, "Analyzing Server Backpressure Mechanisms for Maintaining Latency Guarantees in Multi-threaded Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 35, no. 2, pp. 267–279, feb. 2024. [En línea]. Disponible en: https://ieeexplore.ieee.org/document/10385742