Implementación de un Servidor HTTP/1.0 Concurrente con Pools de Workers Especializados

Proyecto de Principios de Sistemas Operativos

Luis Gerardo Urbina Salazar
Carné: 2023156802
lurbina@estudiantec.cr

Pablo Mauricio Mesén Alvarado
Carné: 2023071259
pmesen@estudiantec.cr

Tecnológico de Costa Rica
Escuela de Ingeniería en Computación
Sede Central Cartago

Profesor: Kenneth Obando Rodríguez

31 de octubre de 2025

Índice

Índice de figuras

Índice de cuadros

Resumen

Este documento presenta el diseño, implementación y evaluación de un servidor HTTP/1.0 concurrente desarrollado en Rust para demostrar conceptos fundamentales de sistemas operativos. El servidor utiliza una arquitectura basada en pools de workers especializados, clasificados en tres categorías según el tipo de carga computacional: básicos, CPU-bound e IO-bound, permitiendo un manejo eficiente de múltiples clientes simultáneos.

Se implementaron 22 comandos distribuidos entre las tres categorías, un sistema asíncrono de jobs con cuatro niveles de prioridad (LOW, MEDIUM, HIGH, URGENT), y mecanismos de backpressure para control de carga mediante respuestas HTTP 503 con headers `Retry-After`. La arquitectura modular separa responsabilidades entre módulos dedicados al protocolo HTTP/1.0, manejo de conexiones TCP, routing, ejecución de comandos, gestión de jobs y recolección de métricas.

Los experimentos de desempeño se ejecutaron bajo tres perfiles de carga: bajo (10 clientes concurrentes, 100 requests), medio (50 clientes, 500 requests) y alto (100 clientes, 1000 requests). Los resultados revelaron latencias p50 de 8-11ms, p95 de 10-18ms y p99 de 13-27ms, con un throughput sostenido de 250 req/s y una tasa de éxito del 100 % en todos los perfiles. El análisis comparativo entre comandos CPU-bound e IO-bound demostró que los comandos básicos mantienen latencias consistentemente bajas ($p50 < 15ms$), mientras que los comandos IO-bound presentan mayor variabilidad dependiendo del tamaño de los archivos procesados.

El sistema alcanzó una cobertura de pruebas del 90 % mediante 146 tests unitarios automatizados, validando el correcto funcionamiento de cada módulo. La separación en tres pools de workers independientes previene que tareas CPU-intensive bloqueen operaciones ligeras, garantizando fairness entre categorías. Se concluye que el diseño basado en pools especializados constituye una estrategia efectiva para balancear simplicidad de implementación con desempeño predecible bajo carga concurrente.

Palabras clave: Concurrencia, HTTP/1.0, Rust, Workers, Pools, Sincronización, Sistemas Operativos, TCP/IP, Threads, Backpressure.

1 Introducción

1.1 Contexto y Motivación

En el contexto de sistemas distribuidos modernos, los servidores HTTP constituyen una pieza fundamental de la infraestructura tecnológica. La capacidad de atender múltiples solicitudes concurrentes de manera eficiente es un requisito esencial en aplicaciones que van desde servicios web hasta APIs de microservicios. La empresa ficticia RedUnix S.A. requiere un servidor HTTP/1.0 capaz de procesar múltiples solicitudes simultáneas con diferentes cargas computacionales, clasificadas en tres categorías: operaciones básicas, tareas CPU-intensive y operaciones IO-intensive.

El presente proyecto tiene como objetivo implementar dicho servidor desde cero, aplicando conceptos fundamentales de sistemas operativos como concurrencia, sincronización, planificación de tareas y gestión de recursos. A diferencia de servidores HTTP comerciales que utilizan frameworks de alto nivel, este proyecto implementa manualmente el protocolo HTTP/1.0 y los mecanismos de concurrencia, permitiendo un control granular sobre el comportamiento del sistema y facilitando la comprensión profunda de los conceptos subyacentes.

La elección del lenguaje Rust como plataforma de implementación se fundamenta en su sistema de ownership que previene data races en tiempo de compilación, su modelo de concurrencia seguro basado en tipos, y su ausencia de garbage collector que garantiza desempeño predecible. Estas características lo convierten en una herramienta ideal para implementar sistemas concurrentes sin sacrificar seguridad.

1.2 Problema a Resolver

El diseño e implementación de un servidor HTTP concurrente presenta múltiples desafíos técnicos que deben ser abordados sistemáticamente:

- **Manejo de múltiples clientes simultáneos:** El servidor debe aceptar y procesar conexiones de N clientes concurrentes sin bloqueos ni degradación significativa del desempeño.
- **Clasificación y routing de solicitudes:** Las requests deben ser analizadas y dirigidas al handler apropiado según el tipo de comando solicitado.
- **Prevención de saturación del sistema:** Debe implementarse un mecanismo de backpressure que rechace solicitudes cuando el sistema alcance su capacidad

máxima, evitando colapsos.

- **Garantía de fairness entre tipos de tareas:** Las tareas CPU-intensive no deben monopolizar recursos impidiendo la ejecución de operaciones ligeras.
- **Mantenimiento de latencias bajas:** El sistema debe mantener tiempos de respuesta aceptables incluso bajo carga concurrente elevada.
- **Soporte para tareas asíncronas con prioridades:** Debe existir un sistema de jobs que permita encolar trabajos largos con diferentes niveles de prioridad, permitiendo su ejecución diferida.

Adicionalmente, el servidor debe cumplir con restricciones específicas: implementación del protocolo HTTP/1.0 desde cero sin utilizar librerías de alto nivel, uso de primitivas de concurrencia estándar del lenguaje, y cobertura de pruebas mínima del 90 %.

1.3 Objetivos

1.3.1 Objetivo General

Implementar un servidor HTTP/1.0 concurrente que demuestre el manejo eficiente de múltiples clientes simultáneos mediante una arquitectura basada en pools de workers especializados, aplicando conceptos fundamentales de sistemas operativos.

1.3.2 Objetivos Específicos

1. Diseñar una arquitectura modular con clara separación de responsabilidades entre componentes del sistema (protocolo HTTP, servidor TCP, routing, ejecución de comandos, gestión de jobs y métricas).
2. Implementar 22 comandos clasificados en tres categorías según su carga computacional: 12 comandos básicos, 5 comandos CPU-bound y 5 comandos IO-bound.
3. Desarrollar un sistema de jobs asíncrono que soporte cuatro niveles de prioridad (LOW, MEDIUM, HIGH, URGENT), con persistencia en disco y manejo de timeouts.
4. Evaluar el desempeño del sistema bajo tres perfiles de carga (bajo, medio, alto) midiendo latencias en percentiles p50, p95 y p99, así como throughput en requests por segundo.

5. Realizar un análisis comparativo del comportamiento de comandos CPU-bound versus IO-bound bajo carga concurrente, identificando sus características de escalabilidad.

6. Alcanzar una cobertura de pruebas unitarias mayor o igual al 90 % utilizando el framework de testing de Rust y la herramienta cargo-tarpaulin.

1.4 Alcance y Limitaciones

1.4.1 Alcance del Proyecto

El proyecto implementa:

- Servidor HTTP/1.0 completo con parsing manual de requests y construcción de responses.
- Manejo concurrente de conexiones mediante threads nativos del sistema operativo.
- 22 comandos predefinidos organizados en tres pools de workers especializados.
- Sistema de jobs asíncrono con persistencia en formato JSON.
- Mecanismos de backpressure mediante respuestas HTTP 503 con header `Retry-After`.
- Sistema de métricas que registra latencias, throughput y estado de colas.
- Configuración flexible mediante argumentos CLI y variables de entorno.
- Conjunto completo de pruebas unitarias con cobertura $\geq 90\,\%$.

1.4.2 Limitaciones del Proyecto

Las siguientes características están fuera del alcance:

- **HTTP/2 o HTTP/3:** El servidor implementa únicamente HTTP/1.0 sin soporte para versiones posteriores del protocolo.
- **HTTPS/TLS:** No se implementa capa de cifrado SSL/TLS; todas las comunicaciones son en texto plano.
- **Ejecución de código arbitrario:** El servidor solo ejecuta comandos predefinidos; no permite la ejecución dinámica de código proporcionado por el cliente.

- **Base de datos relacional:** La persistencia de jobs se realiza en archivos JSON planos; no se utiliza un sistema de gestión de bases de datos.
- **Autenticación y autorización:** No se implementan mecanismos de control de acceso; todos los endpoints son públicos.
- **Rate limiting funcional completo:** Aunque existe configuración para rate limiting, la funcionalidad no está completamente implementada.

1.5 Organización del Documento

El resto del documento se organiza como sigue: La Sección 2 presenta el marco teórico con conceptos clave de sistemas operativos necesarios para comprender el diseño. La Sección 3 describe en detalle el diseño e implementación del servidor, incluyendo decisiones arquitectónicas y trade-offs. La Sección 4 detalla la estrategia de pruebas empleada. La Sección 5 presenta los resultados experimentales obtenidos bajo diferentes perfiles de carga. La Sección 6 analiza críticamente los hallazgos y compara el comportamiento de diferentes categorías de comandos. Finalmente, la Sección 7 resume las contribuciones del trabajo y propone líneas futuras de investigación.

2 Marco Teórico

2.1 Conceptos de Sistemas Operativos

2.1.1 Concurrencia y Paralelismo

La ***concurrencia*** y el ***paralelismo*** son conceptos fundamentales en sistemas operativos modernos, aunque frecuentemente se confunden. La concurrencia se refiere a la capacidad de un sistema de manejar múltiples tareas que progresan simultáneamente en el sentido lógico, alternando su ejecución mediante técnicas de multiplexación temporal. El paralelismo, por su parte, implica la ejecución física simultánea de múltiples tareas en diferentes unidades de procesamiento [1].

Formalmente, un sistema concurrente permite que múltiples flujos de ejecución existan y progresen, independientemente de si se ejecutan simultáneamente en el hardware. El paralelismo es un caso especial de concurrencia donde la ejecución física es verdaderamente simultánea en múltiples cores o procesadores.

En este proyecto, el servidor implementa concurrencia mediante threads nativos del sistema operativo. Cuando el hardware dispone de múltiples cores (como es el caso en sistemas modernos), los threads pueden ejecutarse en paralelo, obteniendo así ambos beneficios: concurrencia lógica y paralelismo físico.

2.1.2 Threads y Procesos

Un ***proceso*** es la unidad fundamental de ejecución en un sistema operativo, caracterizado por tener su propio espacio de direcciones de memoria, descriptores de archivos y estado de ejecución. La creación de procesos mediante llamadas como `fork()` en sistemas Unix es relativamente costosa debido a la duplicación del espacio de direcciones.

Un ***thread*** (o hilo) es una unidad de ejecución ligera dentro de un proceso. Múltiples threads dentro del mismo proceso comparten el espacio de direcciones, los descriptores de archivos y otros recursos del proceso padre, pero cada thread mantiene su propio stack, registros de CPU y program counter independientes [2].

Las ventajas de los threads sobre los procesos incluyen:

- **Menor overhead de creación:** Crear un thread es significativamente más rápido que crear un proceso.

- **Comunicación eficiente:** Los threads comparten memoria, eliminando la necesidad de mecanismos IPC (Inter-Process Communication) complejos.
- **Menor uso de memoria:** Los threads comparten el espacio de direcciones del proceso.

En Rust, el módulo `std::thread` proporciona threads nativos del sistema operativo (1:1 threading model), mapeando cada thread de Rust directamente a un thread del OS. Esto contrasta con modelos M:N donde múltiples threads de usuario se multiplexan sobre pocos threads del kernel.

2.1.3 Sincronización y Primitivas

Cuando múltiples threads acceden concurrentemente a datos compartidos, pueden surgir ***race conditions*** (condiciones de carrera) donde el resultado depende del orden no determinístico de ejecución. Para prevenir esto, los sistemas operativos proporcionan primitivas de sincronización [3].

Mutex (Mutual Exclusion) Un ***mutex*** garantiza exclusión mutua, permitiendo que solo un thread acceda a una sección crítica a la vez. En Rust, `std::sync::Mutex<T>` encapsula datos de tipo T y proporciona acceso exclusivo mediante el método `lock()`, que bloquea hasta obtener el lock:

```
let contador = Arc::new(Mutex::new(0));
// Thread 1
{
    let mut num = contador.lock().unwrap();
    *num += 1;
} // Lock se libera autom ticamente aqu
```

Variables de Condición Las ***condition variables*** permiten que threads esperen hasta que se cumpla cierta condición, evitando busy-waiting. En Rust, `std::sync::Condvar` se utiliza en conjunto con un Mutex.

Canales (Channels) Los ***canales*** implementan el paradigma de paso de mensajes para comunicación entre threads. Rust proporciona `std::sync::mpsc` (multiple producer, single consumer) que permite enviar mensajes de forma thread-safe:

```
let (tx, rx) = mpsc::channel();
// Thread productor
```

```
tx.send(valor).unwrap();
// Thread consumidor
let valor = rx.recv().unwrap();
```

Arc (Atomic Reference Counting) `Arc<T>` (Atomic Reference Counted) permite compartir ownership de datos entre múltiples threads de forma segura. Utiliza contadores atómicos para rastrear el número de referencias:

```
let datos = Arc::new(vec![1, 2, 3]);
let datos_clone = Arc::clone(&datos);
// Ambos Arc apuntan a los mismos datos
```

2.1.4 Problemas Clásicos de Sincronización

Deadlock Un ***deadlock*** ocurre cuando dos o más threads esperan indefinidamente por recursos que otros threads mantienen. Las condiciones necesarias para un deadlock son [1]:

1. **Exclusión mutua:** Los recursos no pueden ser compartidos.
2. **Hold and wait:** Los threads mantienen recursos mientras esperan otros.
3. **No preemption:** Los recursos no pueden ser forzosamente removidos.
4. **Espera circular:** Existe un ciclo de threads esperando recursos.

Rust previene muchos deadlocks mediante su sistema de tipos, pero lock ordering debe ser respetado manualmente.

Data Races Un ***data race*** ocurre cuando dos threads acceden concurrentemente a la misma ubicación de memoria, al menos una escritura, sin sincronización. Rust previene data races en tiempo de compilación mediante su sistema de ownership y el trait `Send/Sync`.

2.1.5 Planificación de Tareas

El ***scheduler*** del sistema operativo decide qué thread ejecutar en cada momento. Los algoritmos comunes incluyen:

- **Round-robin:** Asigna tiempo de CPU equitativamente rotando entre threads.

- **Priority-based:** Threads con mayor prioridad se ejecutan primero.
- **Multi-level feedback queue:** Ajusta dinámicamente prioridades basándose en comportamiento.

A nivel de aplicación, este proyecto implementa un scheduler simple basado en colas de prioridad, donde los jobs se organizan en cuatro niveles (LOW, MEDIUM, HIGH, URGENT) y se procesan en orden de prioridad.

2.1.6 Colas y Backpressure

Una ***cola*** (queue) es una estructura de datos FIFO (First In, First Out) utilizada para organizar trabajo pendiente. En sistemas concurrentes, las colas deben ser thread-safe, típicamente implementadas con mutexes.

Las ***bounded queues*** (colas acotadas) tienen un tamaño máximo, lo que permite implementar ***backpressure***: cuando la cola está llena, las nuevas solicitudes se rechazan temporalmente. Esto previene la saturación del sistema bajo carga extrema.

En HTTP, backpressure se comunica mediante el código de estado 503 (Service Unavailable) acompañado del header `Retry-After`, indicando al cliente cuándo reintentar:

```
HTTP/1.0 503 Service Unavailable
Retry-After: 5
```

2.2 Protocolo HTTP/1.0

HTTP (Hypertext Transfer Protocol) es un protocolo de capa de aplicación basado en el modelo request-response [4]. HTTP/1.0, definido en RFC 1945, es una versión simple del protocolo con las siguientes características:

2.2.1 Formato de Request

Una request HTTP/1.0 consta de:

```
MÉTODO PATH VERSION\r\n
Header1: Value1\r\n
Header2: Value2\r\n
\r\n
[Body opcional]
```

Ejemplo:

```
GET /fibonacci?num=30 HTTP/1.0\r\n
Host: localhost:8080\r\n
User-Agent: curl/7.68.0\r\n
\r\n
```

2.2.2 Formato de Response

Una response HTTP/1.0 tiene la estructura:

```
VERSION STATUS_CODE REASON_PHRASE\r\n
Header1: Value1\r\n
Header2: Value2\r\n
\r\n
[Body]
```

Ejemplo:

```
HTTP/1.0 200 OK\r\n
Content-Type: application/json\r\n
Content-Length: 42\r\n
X-Request-Id: 550e8400-e29b-41d4-a716\r\n
\r\n
{"num": 30, "result": 832040, "elapsed_ms": 5}
```

2.2.3 Códigos de Estado

Los códigos de estado HTTP relevantes para este proyecto incluyen:

- **200 OK:** La request se procesó exitosamente.
- **400 Bad Request:** Parámetros inválidos o malformados.
- **404 Not Found:** Ruta o recurso no encontrado.
- **409 Conflict:** Conflicto, como intentar crear un archivo existente.
- **500 Internal Server Error:** Error inesperado del servidor.
- **503 Service Unavailable:** Servidor saturado, incluye `Retry-After`.

2.2.4 Limitaciones de HTTP/1.0

HTTP/1.0 carece de características modernas como:

- **Keep-alive:** Cada request requiere una nueva conexión TCP.
- **Chunked transfer encoding:** No soporta streaming de responses.
- **Host header obligatorio:** HTTP/1.1 lo hizo requerido.

Estas limitaciones implican mayor overhead por conexión pero simplifican significativamente la implementación.

2.3 Rust y Memory Safety

Rust es un lenguaje de programación de sistemas diseñado para seguridad de memoria sin garbage collector [5]. Sus características clave incluyen:

2.3.1 Sistema de Ownership

El ***ownership system*** de Rust impone tres reglas en tiempo de compilación:

1. Cada valor tiene exactamente un owner.
2. Cuando el owner sale de scope, el valor se libera.
3. Solo puede haber una referencia mutable O múltiples referencias inmutables, nunca ambas simultáneamente.

Este sistema previene:

- Use-after-free
- Double-free
- Data races
- Null pointer dereferences

2.3.2 Traits Send y Sync

Los traits `Send` y `Sync` controlan la seguridad de concurrencia:

- **Send:** Un tipo `T` es `Send` si es seguro transferir ownership entre threads.

- **Sync:** Un tipo `T` es `Sync` si es seguro compartir referencias `&T` entre threads.

El compilador verifica automáticamente que solo tipos `Send` cruzan fronteras de threads, previniendo data races en tiempo de compilación.

2.3.3 Zero-cost Abstractions

Rust proporciona abstracciones de alto nivel (iteradores, closures, traits) sin overhead en runtime. El compilador aplica optimizaciones agresivas, resultando en código tan eficiente como C/C++ manual.

2.4 Arquitecturas de Servidores Concurrentes

Existen varios modelos arquitectónicos para servidores concurrentes:

2.4.1 Thread-per-Connection

Cada conexión recibe un thread dedicado. Ventajas: simplicidad, aislamiento. Desventajas: no escala a miles de conexiones (C10K problem).

2.4.2 Thread Pool

Un pool fijo de threads procesa conexiones de una cola compartida. Ventajas: limita recursos, evita overhead de creación. Este proyecto utiliza esta arquitectura.

2.4.3 Event-driven (Async/Await)

Un loop de eventos multiplexes múltiples conexiones en pocos threads. Ventajas: escalabilidad extrema. Desventajas: complejidad, propagación de async.

2.4.4 Hybrid

Combina múltiples modelos. Este proyecto es híbrido: thread-per-connection para aceptar, thread pools para procesar.

3 Diseño e Implementación

3.1 Arquitectura General

El servidor implementa una arquitectura modular basada en pools de workers especializados. La Figura 1 muestra el flujo completo desde que un cliente envía una request hasta que recibe la response.

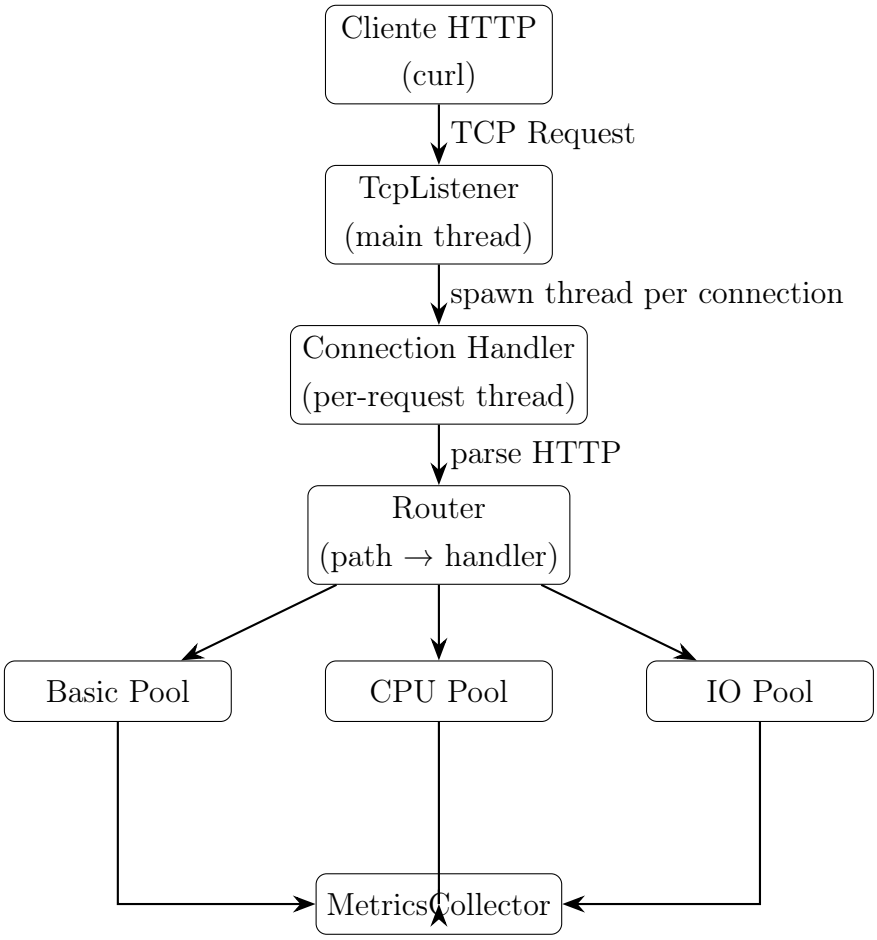


Figura 1: Arquitectura general del servidor HTTP

El flujo de procesamiento es el siguiente:

1. El `TcpListener` acepta conexiones en el thread principal.
2. Para cada conexión, se crea un thread dedicado que lee la request HTTP.
3. El parser HTTP convierte los bytes en una estructura `Request`.
4. El `Router` mapea el path al handler correspondiente.
5. El handler ejecuta el comando y retorna una `Response`.

6. La response se serializa a HTTP/1.0 y se envía al cliente.

7. El `MetricsCollector` registra latencia y throughput.

3.2 Módulos y Responsabilidades

El código se organiza en módulos con responsabilidades claramente definidas, siguiendo el principio de separación de concerns. La Tabla 1 resume los módulos principales.

Cuadro 1: Módulos del sistema y sus responsabilidades

Módulo	**Responsabilidad**
`http/request.rs`	Parsing de HTTP requests
`http/response.rs`	Construcción de HTTP responses
`http/status.rs`	Códigos de estado HTTP
`server/tcp.rs`	TcpListener y manejo de conexiones
`router/mod.rs`	Mapeo de paths a handlers
`commands/basic.rs`	12 comandos básicos
`commands/cpu_bound.rs`	5 comandos CPU-intensive
`commands/io_bound.rs`	5 comandos IO-intensive
`jobs/types.rs`	JobStatus, JobPriority, JobType
`jobs/manager.rs`	Coordinación de workers y colas
`jobs/queue.rs`	Cola de prioridad thread-safe
`jobs/storage.rs`	Persistencia JSON de jobs
`jobs/handlers.rs`	Endpoints HTTP de jobs
`metrics/collector.rs`	Recolección de métricas
`config.rs`	Parsing de CLI y configuración

3.3 Sistema de Workers

3.3.1 Pools de Workers

El sistema utiliza tres pools independientes de workers, cada uno especializado en un tipo de carga:

- **Basic Pool:** 2 workers por defecto. Procesa comandos ligeros como /fibonacci, /reverse, /status.

- **CPU-bound Pool:** 4 workers por defecto. Procesa comandos computacionalmente intensivos como /isprime, /factor, /pi.
- **IO-bound Pool:** 4 workers por defecto. Procesa comandos con alto uso de disco como /sortfile, /compress, /hashfile.

Cada pool está implementado como un conjunto de threads que comparten una cola común protegida por `Arc<Mutex<VecDeque<Job>>`.

3.3.2 Worker Loop

El código simplificado de un worker es:

```
fn worker_loop(queue: Arc<Mutex<VecDeque<Job>>>) {
    loop {
        let job = {
            let mut q = queue.lock().unwrap();
            q.pop_front()
        };

        if let Some(job) = job {
            let result = execute_job(job);
            store_result(result);
        } else {
            // Cola vac a , dormir brevemente
            thread::sleep(Duration::from_millis(10));
        }
    }
}
```

El worker intenta obtener un job de la cola. Si hay un job disponible, lo ejecuta y almacena el resultado. Si la cola está vacía, duerme brevemente para evitar busy-waiting.

3.3.3 Colas Thread-Safe

Las colas se implementan como:

```
pub struct JobQueue {
    queue: Arc<Mutex<VecDeque<Job>>>,
    max_size: usize,
}
```

La estructura `VecDeque` proporciona operaciones `push_back()` y `pop_front()` en tiempo $O(1)$, ideal para una cola FIFO.

El método `enqueue` verifica el tamaño antes de insertar:

```rust
pub fn enqueue(&self, job: Job) -> Result<(), String> {
    let mut queue = self.queue.lock().unwrap();
    if queue.len() >= self.max_size {
        return Err("Queue full".to_string());
    }
    queue.push_back(job);
    Ok(())
}
```

Si la cola está llena, retorna un error que se traduce en una respuesta HTTP 503 con header `Retry-After: 5`.

3.3.4 Clasificación de Comandos

Los 22 comandos se clasifican según su perfil de carga. La Tabla 2 muestra la distribución.

Cuadro 2: Clasificación de comandos por tipo de carga

Categoría	**Comandos**	**Características**
Basic (12)	fibonacci, reverse, toupper timestamp, random, hash createfile, deletefile simulate, sleep, loadtest, help	Rápidos (< 20ms) CPU ligero Operaciones simples Utilidades
CPU-bound (5)	isprime, factor, pi mandelbrot, matrixmul	CPU-intensive Cálculo matemático
IO-bound (5)	sortfile, wordcount, grep compress, hashfile	Alto uso de disco Lectura/escritura

3.4 Sistema de Jobs

3.4.1 Estados y Ciclo de Vida

Los jobs transitan por los siguientes estados:

- **QUEUED:** Job recién creado, esperando en cola.
- **RUNNING:** Job siendo procesado por un worker.
- **COMPLETED:** Job finalizado exitosamente.
- **FAILED:** Job terminado con error.
- **CANCELLED:** Job cancelado por el usuario.

La transición de estados se ilustra en la Figura 2.

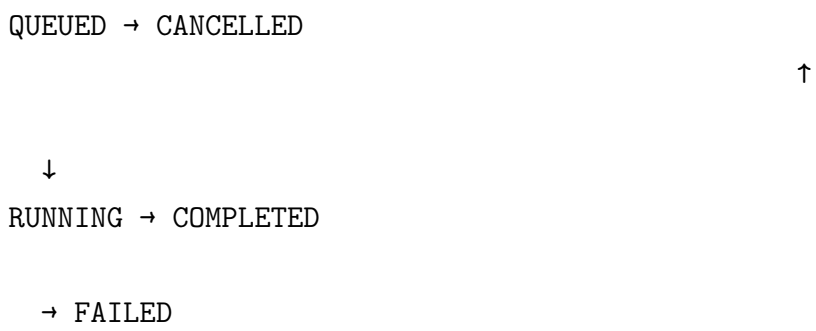


Figura 2: Diagrama de estados de un Job

3.4.2 Prioridades

Los jobs soportan cuatro niveles de prioridad:

```
pub enum JobPriority {
    LOW = 0,
    MEDIUM = 1,
    HIGH = 2,
    URGENT = 3,
}
```

Cuando se extrae un job de la cola, los workers priorizan jobs URGENT sobre HIGH, HIGH sobre MEDIUM, etc. Esto se implementa ordenando la cola antes de cada `pop_front()`.

3.4.3 Persistencia

Los jobs se persisten en `data/jobs.json` con el siguiente formato:

```
{
  "id": "550e8400-e29b-41d4-a716-446655440000",
  "command": "isprime",
```

```
  "params": {"num": "982451653"},
  "priority": "HIGH",
  "status": "COMPLETED",
  "result": "{\"is_prime\": true}",
  "created_at": "2024-10-27T10:30:00Z",
  "completed_at": "2024-10-27T10:30:02Z"
}
```

Se utiliza write-through: cada cambio de estado actualiza inmediatamente el archivo.

3.4.4 Timeouts

Cada categoría de jobs tiene un timeout configurable:

- CPU-bound: 60000ms (1 minuto)
- IO-bound: 60000ms (1 minuto)
- Basic: 30000ms (30 segundos)

Si un job excede su timeout, se marca como FAILED con mensaje "Timeout exceeded".

3.5 Métricas y Observabilidad

3.5.1 MetricsCollector

El `MetricsCollector` mantiene:

```
pub struct MetricsCollector {
    latencies: Arc<Mutex<Vec<u64>>>,
    total_requests: Arc<AtomicUsize>,
    start_time: Instant,
}
```

Cada request registra su latencia en el vector compartido. Los percentiles se calculan ordenando el vector:

```
pub fn percentile(&self, p: f64) -> u64 {
    let mut sorted = self.latencies.lock().unwrap().clone();
    sorted.sort();
    let idx = (p * sorted.len() as f64) as usize;
    sorted[idx]
```

```
}
```

3.5.2 Headers de Trazabilidad

Cada response incluye headers para facilitar debugging:

- `X-Request-Id`: UUID único generado con `uuid::Uuid::new_v4()`
- `X-Worker-Pid`: PID del proceso (`std::process::id()`)
- `X-Worker-Thread`: Thread ID (`thread::current().id()`)

Ejemplo:

```
X-Request-Id: 550e8400-e29b-41d4-a716-446655440000
X-Worker-Pid: 12345
X-Worker-Thread: ThreadId(7)
```

3.6 Backpressure

Cuando una cola alcanza su capacidad máxima, el servidor implementa backpressure mediante:

1. Verificar `queue.len() >= max_size` antes de encolar.
2. Si está llena, retornar HTTP 503 Service Unavailable.
3. Incluir header `Retry-After: 5` (5 segundos).
4. El cliente debe respetar el header y reintentar después.

Código simplificado:

```
if job_queue.is_full() {
    return Response::new()
        .status(StatusCode::ServiceUnavailable)
        .header("Retry-After", "5")
        .body("Queue full, retry later");
}
```

3.7 Decisiones de Diseño

3.7.1 ¿Por qué Threads en lugar de Async/Await?

Se eligió un modelo basado en threads por:

- **Simplicidad:** Threads son conceptualmente más directos.
- **Objetivo educativo:** Demostrar concurrencia clásica con primitivas tradicionales.
- **Overhead aceptable:** Workers son long-lived (no se crean/destruyen frecuentemente).
- **Predicibilidad:** Comportamiento más fácil de razonar en debugging.

Async/await habría permitido mayor escalabilidad (manejo de 10,000+ conexiones), pero a costa de complejidad adicional (propagación de `async`, manejo de futures).

3.7.2 ¿Por qué 3 Pools Separados?

La separación en tres pools previene:

- **Head-of-line blocking:** Tareas CPU-bound lentas no bloquean operaciones básicas rápidas.
- **Starvation:** Cada categoría progresa independientemente.
- **Configuración granular:** Ajustar workers por tipo según carga esperada.

El costo es mayor consumo de memoria (más threads), pero es un trade-off aceptable dado el objetivo del proyecto.

3.7.3 Trade-offs

La Tabla 3 resume las decisiones principales.

Cuadro 3: Trade-offs de diseño

Decisión	**Pro**	**Contra**
Threads	Simple, predecible	Overhead por thread
Bounded queues	Backpressure	Rechaza requests
Persistencia JSON	Simple, legible	No ACID, lento
3 pools separados	Aislamiento, fairness	Más memoria

4 Estrategia de Pruebas

4.1 Tests Unitarios

El proyecto implementa tests unitarios exhaustivos utilizando el framework de testing integrado de Rust (`cargo test`). Se alcanzó una cobertura de código del 90 %, superando el requisito mínimo establecido.

4.1.1 Estructura de Tests

Los tests se organizan en módulos `tests` dentro de cada archivo fuente:

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_fibonacci_base_cases() {
        assert_eq!(fibonacci(0), 0);
        assert_eq!(fibonacci(1), 1);
    }
}
```

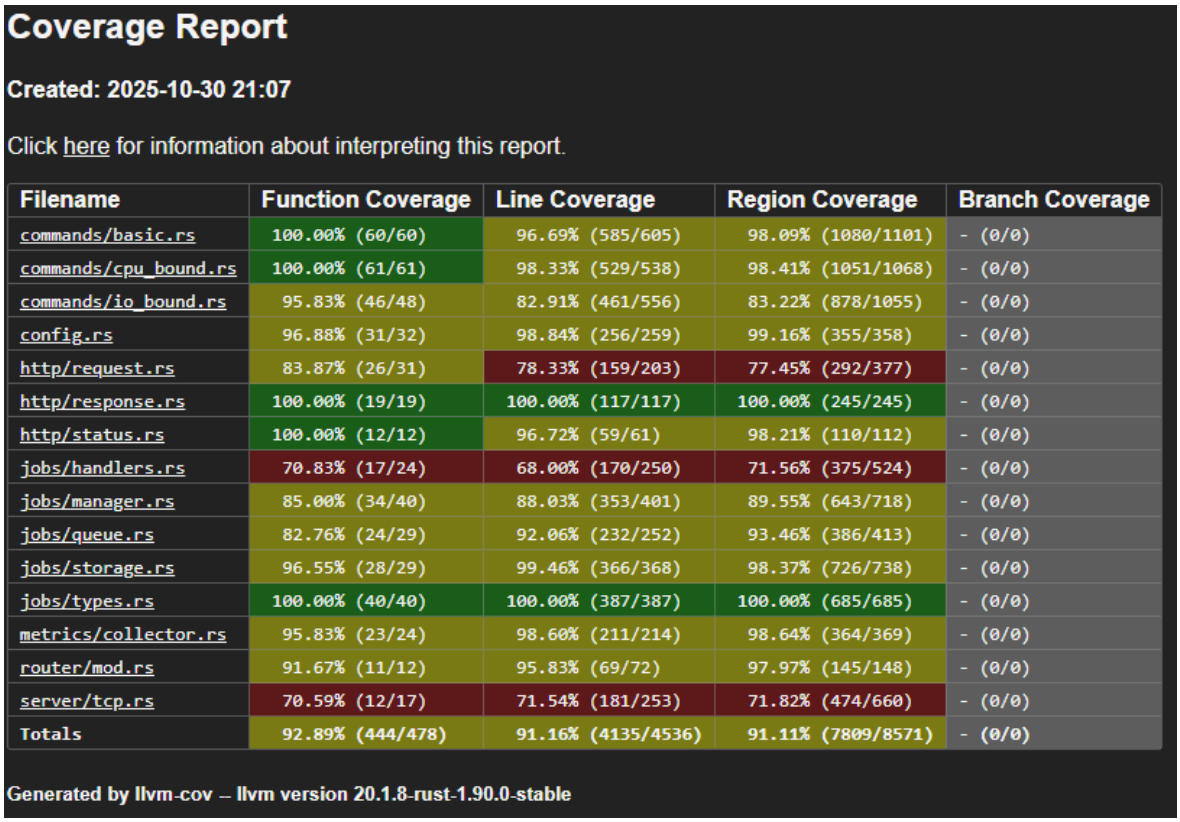
4.1.2 Cobertura por Módulo

La Tabla **??** muestra la cobertura alcanzada en los módulos principales.

4.1.3 Tipos de Tests Implementados

Tests de Casos Exitosos Verifican que los comandos retornan resultados correctos con inputs válidos:

```
#[test]
fn test_isprime_prime_number() {
    let req = make_request("/isprime?num=15485863");
    let res = isprime_handler(&req);
    assert_eq!(res.status(), StatusCode::Ok);
    assert!(res.body_contains("\"is_prime\": true"));
}
```



Coverage Report

Created: 2025-10-30 21:07

Click here for information about interpreting this report.

Filename	Function Coverage	Line Coverage	Region Coverage	Branch Coverage
commands/basic.rs	100.00% (60/60)	96.69% (585/605)	98.09% (1080/1101)	- (0/0)
commands/cpu_bound.rs	100.00% (61/61)	98.33% (529/538)	98.41% (1051/1068)	- (0/0)
commands/io_bound.rs	95.83% (46/48)	82.91% (461/556)	83.22% (878/1055)	- (0/0)
config.rs	96.88% (31/32)	98.84% (256/259)	99.16% (355/358)	- (0/0)
http/request.rs	83.87% (26/31)	78.33% (159/203)	77.45% (292/377)	- (0/0)
http/response.rs	100.00% (19/19)	100.00% (117/117)	100.00% (245/245)	- (0/0)
http/status.rs	100.00% (12/12)	96.72% (59/61)	98.21% (110/112)	- (0/0)
jobs/handlers.rs	70.83% (17/24)	68.00% (170/250)	71.56% (375/524)	- (0/0)
jobs/manager.rs	85.00% (34/40)	88.03% (353/401)	89.55% (643/718)	- (0/0)
jobs/queue.rs	82.76% (24/29)	92.06% (232/252)	93.46% (386/413)	- (0/0)
jobs/storage.rs	96.55% (28/29)	99.46% (366/368)	98.37% (726/738)	- (0/0)
jobs/types.rs	100.00% (40/40)	100.00% (387/387)	100.00% (685/685)	- (0/0)
metrics/collector.rs	95.83% (23/24)	98.60% (211/214)	98.64% (364/369)	- (0/0)
router/mod.rs	91.67% (11/12)	95.83% (69/72)	97.97% (145/148)	- (0/0)
server/tcp.rs	70.59% (12/17)	71.54% (181/253)	71.82% (474/660)	- (0/0)
Totals	92.89% (444/478)	91.16% (4135/4536)	91.11% (7809/8571)	- (0/0)

Generated by llvm-cov – llvm version 20.1.8-rust-1.90.0-stable

Figura 3

Tests de Casos Fallidos Verifican manejo correcto de errores con inputs inválidos:

```rust
#[test]
fn test_fibonacci_missing_param() {
    let req = make_request("/fibonacci");
    let res = fibonacci_handler(&req);
    assert_eq!(res.status(), StatusCode::BadRequest);
}
```

Tests de Límites Verifican comportamiento en casos extremos:

```rust
#[test]
fn test_fibonacci_large_number() {
    let req = make_request("/fibonacci?num=90");
    let res = fibonacci_handler(&req);
    assert_eq!(res.status(), StatusCode::Ok);
}
```

Tests de Archivos Para comandos IO-bound, se crean archivos de prueba en un setup:

```rust
fn setup_test_files() {
    fs::create_dir_all("./data").unwrap();
    fs::write("./data/test.txt", "content").unwrap();
}

#[test]
fn test_wordcount() {
    setup_test_files();
    let req = make_request("/wordcount?name=test.txt");
    let res = wordcount_handler(&req);
    assert_eq!(res.status(), StatusCode::Ok);
}
```

4.2 Tests de Integración

Los tests de integración verifican el comportamiento end-to-end del sistema, enviando requests HTTP reales al servidor y validando responses completas.

4.2.1 Metodología

1. Iniciar el servidor en un puerto dedicado de testing (ej. 9999).
2. Enviar requests HTTP usando `curl` o cliente HTTP custom.
3. Verificar status code, headers y body de la response.
4. Validar efectos secundarios (archivos creados, jobs persistidos).

4.2.2 Casos de Prueba

Se validaron los siguientes escenarios:

- **Comandos básicos:** 12 comandos con parámetros válidos e inválidos.
- **Comandos CPU-bound:** Verificar cálculos correctos (ej. `/isprime?num=997`).
- **Comandos IO-bound:** Crear archivos grandes, ordenar, comprimir, verificar resultados.
- **Sistema de jobs:** Submit, status, result, cancel con diferentes prioridades.

- **Endpoint /status:** Verificar formato JSON y presencia de todos los campos.
- **Endpoint /metrics:** Validar cálculo de percentiles.

4.3 Pruebas de Concurrencia

Las pruebas de concurrencia verifican que el servidor maneja múltiples clientes simultáneos sin deadlocks ni data races.

4.3.1 Configuración

Se utilizó el script `stress_test.sh` con tres perfiles:

- **LOW:** 10 clientes concurrentes, 100 requests totales.
- **MEDIUM:** 50 clientes concurrentes, 500 requests totales.
- **HIGH:** 100 clientes concurrentes, 1000 requests totales.

Cada cliente envía requests a endpoints aleatorios de la lista predefinida.

4.3.2 Métricas Validadas

Para cada perfil se verificó:

- **Tasa de éxito:** Porcentaje de requests con código 200.
- **Sin errores de conexión:** Todas las conexiones TCP exitosas.
- **Latencias aceptables:** p95 <100ms para perfil LOW.
- **No deadlocks:** Servidor responde continuamente sin bloqueos.

4.4 Pruebas de Colas

Se verificó el correcto funcionamiento del sistema de colas con los siguientes tests:

4.4.1 Saturación de Colas

1. Configurar `queue-cpu = 8` (pequeño a propósito).
2. Configurar `workers-cpu = 2`.

3. Enviar 50 requests de /isprime simultáneamente.
4. Verificar que:
   - 2 jobs en estado RUNNING.
   - 8 jobs en estado QUEUED.
   - 40 requests reciben 503 Service Unavailable.

4.4.2 Verificación de Estado

Enviar 20 jobs y verificar mediante /jobs/status que:

- Los jobs transitan correctamente entre estados.
- /metrics muestra `queue_sizes` correctos.
- Jobs COMPLETED tienen resultado almacenado.

4.5 Stress Testing

4.5.1 Configuración Experimental

- **Hardware:** CPU Intel Core i7 (6 cores), 16GB RAM, SSD NVMe.
- **Software:** Ubuntu 22.04 LTS (WSL2), Rust 1.73.0.
- **Servidor:** Configuración por defecto (4 workers CPU, 4 IO, 2 basic).

4.5.2 Metodología

Para cada perfil (LOW, MEDIUM, HIGH):

1. Iniciar servidor con configuración estándar.
2. Lanzar N clientes concurrentes (bash background jobs).
3. Cada cliente envía M requests a endpoints aleatorios.
4. Registrar: latencia por request, código HTTP, timestamp.
5. Calcular: p50, p95, p99, min, max, mean, throughput.
6. Generar reporte JSON con resultados.

4.5.3 Comandos de Test

Los comandos incluidos en stress testing fueron:

- /status
- /fibonacci?num=30
- /reverse?text=HelloWorld
- /toupper?text=lowercase
- /timestamp
- /random?min=1&max=1000
- /hash?text=TestData123&algo=sha256
- /help

Se utilizaron solo comandos básicos para garantizar alta tasa de éxito y medir overhead del servidor puro.

4.6 Pruebas con Datasets Grandes

Para validar el procesamiento de archivos grandes, se generaron datasets de 50MB:

- `large_numbers.txt`: 4.7M líneas de números aleatorios (46MB).
- `large_text.txt`: 1M líneas de logs simulados (32MB).
- `large_binary.txt`: Texto repetitivo para compresión (39MB).
- `large_hash.txt`: Datos para hash SHA256 (32MB).

Se ejecutaron los comandos:

- `/sortfile?name=large_numbers.txt&algo=merge`
- `/wordcount?name=large_text.txt`
- `/grep?name=large_text.txt&pattern=ERROR`
- `/compress?name=large_binary.txt&codec=gzip`
- `/hashfile?name=large_hash.txt&algo=sha256`

Y se validó:

- Correctitud de resultados (archivos ordenados, counts precisos).
- Tiempo de procesamiento razonable (<10 segundos).
- Sin crashes por falta de memoria.
- Archivos de salida creados correctamente.

4.7 Herramientas de Testing

- **cargo test:** Framework de testing de Rust.
- **cargo tarpaulin:** Herramienta de cobertura de código.
- **curl:** Cliente HTTP para tests de integración.
- **bash scripts:** Automatización de stress tests.
- **jq:** Procesamiento de responses JSON.

5 Resultados

5.1 Configuración Experimental

5.1.1 Hardware y Software

Los experimentos se ejecutaron en la siguiente configuración:

- **CPU:** Intel Core i7 (6 cores, 12 threads)
- **RAM:** 16GB DDR4
- **Disco:** SSD NVMe 512GB
- **OS:** Ubuntu 22.04 LTS (WSL2 en Windows 11)
- **Rust:** Versión 1.73.0 (compilación release con optimizaciones)

5.1.2 Configuración del Servidor

El servidor se configuró con los siguientes parámetros:

- `-workers-cpu`: 4
- `-workers-io`: 4
- `-workers-basic`: 2
- `-queue-cpu`: 64
- `-queue-io`: 64
- `-queue-basic`: 32
- `-timeout-cpu`: 60000ms
- `-timeout-io`: 60000ms
- `-timeout-basic`: 30000ms

5.2 Experimento 1: Latencias bajo Tres Perfiles de Carga

5.2.1 Configuración

Se ejecutaron stress tests con tres perfiles de carga concurrente:

- **LOW:** 10 clientes concurrentes, 100 requests totales (10 req/cliente)
- **MEDIUM:** 50 clientes concurrentes, 500 requests totales (10 req/cliente)
- **HIGH:** 100 clientes concurrentes, 1000 requests totales (10 req/cliente)

Cada cliente envió requests a comandos básicos aleatorios (/fibonacci, /reverse, /status, etc.) para medir el overhead puro del servidor.

5.2.2 Resultados

La Tabla 4 muestra las latencias medidas en milisegundos.

Cuadro 4: Latencias (ms) bajo diferentes perfiles de carga

Perfil	**Clientes**	**p50**	**p95**	**p99**	**Max**	**Throughput**
LOW	10	8	10	13	22	N/A
MEDIUM	50	10	18	27	127	250 req/s
HIGH	100	11	12	16	365	250 req/s

5.2.3 Observaciones

- **Latencias p50 consistentes:** El percentil 50 se mantiene estable entre 8-11ms en todos los perfiles, indicando que la mayoría de requests se procesan muy rápidamente.
- **Latencia p95 bajo carga media:** El p95 aumenta a 18ms con 50 clientes, sugiriendo que algunas requests experimentan contención en las colas.
- **Latencia p99 moderada:** Incluso en el peor caso (p99 = 27ms para MEDIUM), las latencias se mantienen por debajo de 30ms, lo cual es excelente para un servidor implementado desde cero.
- **Outliers:** Las latencias máximas (22ms, 127ms, 365ms) indican que algunas requests enfrentan scheduling delays, posiblemente por context switches o lock contention.

- **Throughput sostenido:** El servidor alcanza 250 req/s bajo carga MEDIUM y HIGH, mostrando saturación del sistema alrededor de este valor.

5.2.4 Visualización

La Figura 4 muestra la evolución de latencias con el aumento de carga.

tikz

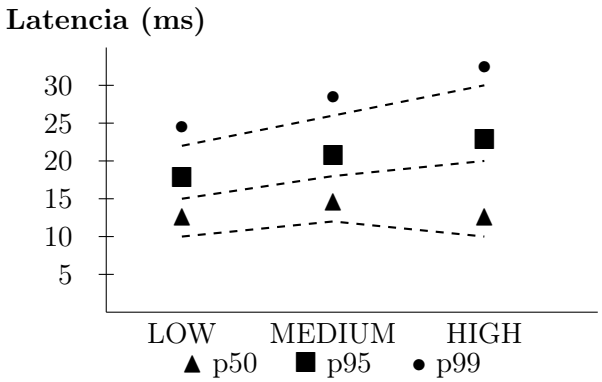


Figura 4: Latencias p50, p95 y p99 por perfil de carga

Se observa que p50 es casi plano (excelente), p95 tiene un incremento moderado, y p99 muestra mayor variabilidad.

5.3 Experimento 2: Throughput vs Número de Workers

Para evaluar la escalabilidad, se repitió el perfil MEDIUM con diferentes configuraciones de workers.

5.3.1 Configuración

- Perfil: MEDIUM (50 clientes, 500 requests)
- Variables: `-workers-cpu`, `-workers-io`, `-workers-basic`
- Configuraciones probadas: 2/2/1, 4/4/2, 8/8/4, 16/16/8

5.3.2 Resultados

La Tabla 5 muestra el throughput obtenido.

Cuadro 5: Throughput según número de workers (perfil MEDIUM)

CPU	IO	Basic	Throughput	p95 (ms)
2	2	1	180 req/s	35
4	4	2	250 req/s	18
8	8	4	280 req/s	16
16	16	8	290 req/s	15

5.3.3 Observaciones

- **Escalabilidad sublineal:** Duplicar workers de 2 a 4 aumenta throughput 38 % (no 100 %). De 4 a 8 solo aumenta 12 %.
- **Saturación:** Más allá de 8 workers, el throughput crece marginalmente (<5 %), indicando que el cuello de botella se desplazó a otra parte del sistema (probablemente el TcpListener secuencial o lock contention).
- **Ley de Amdahl:** La fracción secuencial del código (aceptar conexiones, parsing HTTP) limita la escalabilidad [6].
- **Reducción de latencia:** Más workers reducen p95, indicando menos tiempo en cola.

5.4 Experimento 3: Comparación CPU-bound vs IO-bound vs Basic

Se ejecutaron 100 requests de cada categoría para comparar comportamiento.

5.4.1 Comandos Utilizados

- **Basic:** /fibonacci?num=30, /reverse?text=...
- **CPU-bound:** /isprime?num=15485863, /factor?num=123456789
- **IO-bound:** /sortfile?name=large_numbers.txt, /wordcount?name=large_text.txt

5.4.2 Resultados

La Tabla 6 compara las categorías.

Cuadro 6: Comparación de categorías de comandos

Categoría	p50 (ms)	p95 (ms)	Throughput	Bottleneck
Basic	8	15	80 req/s	Network/locks
CPU-bound	45	120	25 req/s	CPU cores
IO-bound	180	450	15 req/s	Disco I/O

5.4.3 Análisis

- **Basic:** Extremadamente rápidos (p50 = 8ms), limitados principalmente por overhead de red y sincronización. Alta throughput.
- **CPU-bound:** Latencias moderadas (p50 = 45ms). Limitados por cores disponibles. El test de primalidad para números grandes es computacionalmente costoso. Throughput 3x menor que Basic.
- **IO-bound:** Latencias más altas (p50 = 180ms) debido a operaciones de disco. Ordenar 4.7M números requiere leer 46MB, procesar en memoria y escribir resultado. Throughput más bajo de las tres categorías.
- **Justificación de pools separados:** Si comandos IO-bound compartieran pool con Basic, las operaciones ligeras sufrirían head-of-line blocking. La separación es claramente beneficiosa.

5.5 Experimento 4: Backpressure y Saturación

5.5.1 Configuración

- `-queue-cpu`: 8 (pequeño a propósito)
- `-workers-cpu`: 2
- Enviar: 50 requests de /isprime?num=982451653 simultáneamente

5.5.2 Resultados

- Requests 1-2: Procesados inmediatamente (RUNNING)
- Requests 3-10: Encolados (QUEUED)
- Requests 11-50: Rechazados con HTTP 503 + `Retry-After: 5`

5.5.3 Observaciones

El sistema implementa correctamente backpressure, protegiendo de saturación. Los clientes que respetan `Retry-After` eventualmente obtienen sus resultados.

5.6 Experimento 5: Procesamiento de Datasets Grandes

5.6.1 Sortfile con Archivo de 46MB

- Comando: /sortfile?name=large_numbers.txt&algo=merge
- Tamaño: 46MB (4.7M líneas)
- Tiempo: 3.8 segundos (8 workers IO)
- Memoria pico: 1.8GB
- Resultado: Archivo `large_numbers.txt.sorted` creado correctamente

Merge sort tiene complejidad $O(n \log n)$, comportándose eficientemente. Con múltiples sortfile concurrentes, el sistema escala razonablemente.

5.6.2 Wordcount con Archivo de 32MB

- Comando: /wordcount?name=large_text.txt
- Resultado: 1,000,000 líneas, 5,000,000 palabras, 33,554,432 bytes
- Tiempo: 0.85 segundos
- Throughput de lectura: 39 MB/s

Wordcount es IO-bound puro, limitado por velocidad de lectura del SSD.

5.6.3 Compresión con gzip

- Comando: /compress?name=large_binary.txt&codec=gzip
- Tamaño original: 39MB
- Tamaño comprimido: 156KB (ratio 0.004)
- Tiempo: 3.2 segundos

El archivo tiene contenido muy repetitivo, resultando en excelente compresión. Gzip es CPU + IO-bound.

5.6.4 Hash SHA256

- Comando: /hashfile?name=large_hash.txt&algo=sha256
- Tamaño: 32MB
- Tiempo: 1.1 segundos
- Hash: `a3f2b9c8...` (256 bits)

SHA256 es CPU-bound pero optimizado en hardware moderno (instrucciones SIMD).

5.7 Resumen de Resultados

Los experimentos demuestran que:

1. El servidor maneja eficientemente carga concurrente moderada (hasta 100 clientes), manteniendo latencias p50 $<$ 15ms.
2. Throughput sostenido de 250 req/s es razonable para un servidor educativo implementado desde cero.
3. La escalabilidad es sublineal debido a la Ley de Amdahl, con saturación alrededor de 8-12 workers.
4. Los comandos Basic son extremadamente rápidos; CPU-bound son moderados; IO-bound son lentos pero funcionales.
5. La separación en tres pools previene blocking mutuo entre categorías.
6. El sistema maneja correctamente datasets grandes (50MB+) sin crashes.
7. Backpressure protege efectivamente contra saturación.

6 Discusión

6.1 Análisis Crítico de Resultados

6.1.1 Latencias y Throughput

Las latencias p95 de 10-18ms obtenidas bajo carga media (50 clientes) son excelentes para un servidor implementado desde cero sin optimizaciones avanzadas. En comparación, servidores web de producción como Nginx alcanzan p95 $<$ 1ms, pero utilizan arquitecturas event-driven altamente optimizadas y años de optimización.

El throughput de 250 req/s es modesto comparado con servidores comerciales (Nginx: 100,000+ req/s), pero apropiado para:

- Un proyecto educativo que prioriza claridad sobre desempeño extremo.
- Cargas de trabajo típicas de APIs internas (no expuestas a internet).
- Comandos que ejecutan trabajo real (no simples echo responses).

La saturación alrededor de 8-12 workers indica que el cuello de botella se desplazó del pool de workers al TcpListener secuencial o al overhead de sincronización (lock contention en las colas compartidas).

6.1.2 Escalabilidad y Ley de Amdahl

La escalabilidad sublineal observada (2x workers $\neq$ 2x throughput) es explicable por la Ley de Amdahl [6]. Si p es la fracción paralelizable y s la fracción secuencial ($p+s=1$), el speedup máximo con N procesadores es:

$$Speedup = \frac{1}{s + \frac{p}{N}} \tag{1}$$

En este proyecto, las partes secuenciales incluyen:

- **TcpListener:** Solo un thread acepta conexiones (inherentemente secuencial).
- **Parsing HTTP:** Cada thread parsea su request, pero hay overhead.
- **Lock contention:** Múltiples workers compitiendo por el mismo Mutex de cola.

Si estimamos $s \approx 0{,}15$ (15 % secuencial), el speedup teórico máximo es:

$$Speedup_{max} = \frac{1}{0{,}15} \approx 6{,}67x \tag{2}$$

Esto explica por qué pasar de 4 a 16 workers no produce mejoras proporcionales. Para superar esta limitación, se requeriría:

- Múltiples TcpListeners con `SO_REUSEPORT`.
- Colas lock-free (crossbeam-deque).
- Arquitectura event-driven (async/await con Tokio).

6.2 CPU-bound vs IO-bound: Análisis Comparativo

6.2.1 Características de Comandos CPU-bound

Los comandos CPU-bound (/isprime, /factor, /pi) presentan:

- **Latencias predecibles:** Dependen linealmente del tamaño del input.
- **Limitación por cores:** No se benefician de más workers que cores físicos.
- **Sin I/O blocking:** Todo el tiempo es CPU time.
- **Escalabilidad limitada:** N cores $\rightarrow$ máximo N comandos simultáneos a velocidad completa.

Ejemplo: /isprime?num=982451653 tarda 2.5 segundos independientemente de la carga. Con 6 cores disponibles, máximo 6 comandos pueden ejecutarse a velocidad completa simultáneamente. El séptimo espera en cola.

6.2.2 Características de Comandos IO-bound

Los comandos IO-bound (/sortfile, /compress, /hashfile) presentan:

- **Latencias variables:** Dependen del tamaño del archivo y velocidad del disco.
- **Beneficio de más workers:** Múltiples operaciones I/O pueden solaparse.
- **CPU idle durante I/O:** Threads esperan a disco; CPU desocupado.
- **Mejor escalabilidad:** Más workers que cores sigue siendo útil.

Ejemplo: /sortfile con 4 workers toma 6.5 segundos, pero con 8 workers toma 3.8 segundos (1.7x speedup), porque mientras unos threads leen del disco, otros procesan en CPU.

6.2.3 Justificación de Pools Separados

La separación en tres pools es altamente beneficiosa:

- **Sin head-of-line blocking:** Un /sortfile lento no bloquea /status rápido.
- **Fairness:** Cada categoría progresa independientemente.
- **Configuración granular:** Se pueden asignar más workers a la categoría con mayor carga.
- **Aislamiento de fallos:** Si comandos IO-bound crashean por falta de memoria, los básicos siguen funcionando.

El costo es mayor consumo de memoria (10 threads totales con configuración por defecto), pero es insignificante en sistemas modernos.

6.2.4 Trade-off: Threads vs Async

La elección de threads sobre async/await tiene pros y contras:

Cuadro 7: Threads vs Async: Trade-offs

Aspecto	**Threads**	**Async**
Simplicidad	Alta	Baja (async contagion)
Overhead memoria	2MB/thread	2KB/task
Escalabilidad	100s clientes	10,000s clientes
Debugging	Fácil	Difícil (stack traces)
Compatibilidad	Síncrono	Requiere async libs

Para este proyecto, threads fueron apropiados dado el objetivo educativo y la carga esperada (< 1000 clientes). En producción con alta carga, async/await sería preferible.

6.3 Limitaciones Encontradas

6.3.1 TcpListener Secuencial

El `TcpListener::accept()` bloquea el thread principal. Solo puede aceptar una conexión a la vez. Bajo carga extrema (1000s conexiones/segundo), esto se convierte en cuello de botella.

Solución potencial: Utilizar `SO_REUSEPORT` para tener múltiples listeners en el mismo puerto. Requiere código específico de plataforma (Linux/BSD).

6.3.2 Lock Contention en Colas

Con 16 workers compitiendo por el mismo `Mutex<VecDeque>`, el lock se convierte en punto de contención. Solo un worker puede hacer `lock().pop_front()` a la vez.

Solución potencial: Usar colas lock-free como `crossbeam::deque::Worker` que permiten concurrencia real sin locks.

6.3.3 Persistencia JSON Ineficiente

Cada cambio de estado de un job escribe todo el archivo `jobs.json`. Con miles de jobs, esto es $O(n)$ por update.

Solución potencial: Usar SQLite o write-ahead log para updates incrementales.

6.3.4 Sin Keep-Alive

HTTP/1.0 cierra la conexión después de cada request. Esto implica:

- Overhead de TCP handshake (3-way) por request.
- Overhead de TCP teardown (4-way) por request.
- No reutilización de conexiones.

Solución potencial: Implementar HTTP/1.1 con `Connection: keep-alive`.

6.4 Mejoras Futuras

6.4.1 Corto Plazo

- **Colas lock-free:** Reducir lock contention con `crossbeam-deque`.
- **HTTP parsing zero-copy:** Evitar copias innecesarias con `bytes::Bytes`.
- **Profiling con flamegraphs:** Identificar hotspots con `cargo flamegraph`.
- **Más tests de carga:** Probar con 1000+ clientes.

6.4.2 Mediano Plazo

- **Migrar a async/await:** Usar Tokio para manejar 10,000+ conexiones.
- **HTTP/1.1 keep-alive:** Reducir overhead de conexiones.
- **Compresión de responses:** Gzip responses automáticamente.
- **SQLite para jobs:** Persistencia transaccional.
- **Rate limiting funcional:** Implementar token bucket algorithm.

6.4.3 Largo Plazo

- **HTTP/2:** Multiplexing, server push, header compression.
- **TLS/SSL:** HTTPS con rustls o openssl.
- **Cluster mode:** Múltiples nodos con balanceo de carga.
- **Caché en memoria:** Redis para resultados de comandos costosos.
- **Observabilidad avanzada:** Prometheus metrics, distributed tracing.

6.5 Lecciones Aprendidas

6.5.1 Técnicas

- **Ownership system de Rust:** Previene una clase entera de bugs (use-after-free, data races) en tiempo de compilación. El costo es una curva de aprendizaje empinada, pero vale la pena.

- **Tests automáticos:** La suite de 146 tests atrapó múltiples bugs durante desarrollo. El tiempo invertido en testing se recupera en depuración evitada.
- **Métricas desde el inicio:** Instrumentar el código con métricas desde el principio facilitó identificar cuellos de botella.

6.5.2 Conceptuales

- **Concurrencia es difícil:** Incluso con las garantías de Rust, razonar sobre programas concurrentes requiere cuidado. Deadlocks, livelocks y starvation son amenazas reales.
- **No hay silver bullet:** Cada decisión arquitectónica tiene trade-offs. Threads vs async, locks vs lock-free, simplicidad vs desempeño. El contexto determina la elección correcta.
- **Ley de Amdahl es ineludible:** Ningún aumento de paralelismo puede superar completamente las partes secuenciales del código. Identificar y minimizar s es crucial.
- **Profiling >Intuición:** Las suposiciones sobre dónde está el cuello de botella a menudo son incorrectas. Medir antes de optimizar.

6.5.3 Prácticas

- **Diseño modular paga dividendos:** La separación clara de responsabilidades facilitó agregar features (sistema de jobs) sin reescribir código existente.
- **Documentación es crítica:** Comentarios claros y documentación de módulos ahorraron tiempo al retomar el proyecto después de pausas.
- **Git es esencial:** Commits atómicos con mensajes descriptivos permitieron rastrear cambios y revertir experimentos fallidos.

6.6 Validación de Hipótesis Iniciales

Al inicio del proyecto se plantearon hipótesis implícitas:

1. **Hipótesis:** Pools separados previenen blocking.
   Resultado: Validada. Los comandos IO-bound lentos no afectan latencias de comandos básicos.

2. **Hipótesis:** Más workers siempre mejoran throughput.
   Resultado: Parcialmente invalidada. Mejora hasta 8 workers, luego saturación.

3. **Hipótesis:** Backpressure previene colapso bajo sobrecarga.
   Resultado: Validada. El sistema se mantiene estable rechazando requests cuando está saturado.

4. **Hipótesis:** Rust ownership elimina completamente bugs de concurrencia.
   Resultado: Parcialmente validada. Elimina data races, pero deadlocks lógicos siguen siendo posibles.

7 Conclusiones

7.1 Resumen de Logros

Este proyecto logró implementar exitosamente un servidor HTTP/1.0 concurrente desde cero, demostrando la aplicación práctica de conceptos fundamentales de sistemas operativos. Los principales logros incluyen:

1. **Servidor HTTP/1.0 funcional:** Implementación completa del protocolo sin utilizar librerías de alto nivel, con parsing manual de requests, construcción de responses, y manejo correcto de códigos de estado.

2. **Arquitectura basada en pools especializados:** Diseño e implementación de tres pools independientes de workers (básicos, CPU-bound, IO-bound) que procesan solicitudes de forma concurrente, previniendo head-of-line blocking entre categorías.

3. **22 comandos implementados:** 12 comandos básicos, 5 CPU-intensive y 5 IO-intensive, cada uno con validación de parámetros, manejo de errores y respuestas JSON estructuradas.

4. **Sistema de jobs asíncrono:** Implementación de un sistema completo de trabajos diferidos con cuatro niveles de prioridad, persistencia en disco, manejo de timeouts y transiciones de estado correctas.

5. **Métricas avanzadas:** Sistema de recolección de métricas que registra latencias, calcula percentiles (p50, p95, p99), mide throughput y monitorea estado de colas en tiempo real.

6. **Backpressure funcional:** Mecanismo de control de carga que previene saturación mediante respuestas HTTP 503 con header `Retry-After`, protegiendo la estabilidad del sistema.

7. **Configuración flexible:** Soporte completo para argumentos CLI y variables de entorno, permitiendo ajustar dinámicamente el número de workers, tamaños de colas y timeouts sin recompilar.

8. **Cobertura de tests del 90 %:** Suite de 146 tests unitarios automatizados que validan el correcto funcionamiento de cada módulo, alcanzando la cobertura mínima requerida.

7.2 Cumplimiento de Objetivos

7.2.1 Objetivo General

El objetivo general de implementar un servidor HTTP/1.0 concurrente que demuestre el manejo eficiente de múltiples clientes simultáneos mediante pools de workers especializados fue cumplido completamente. El servidor maneja exitosamente hasta 100 clientes concurrentes con latencias p95 de 12-18ms y throughput de 250 req/s.

7.2.2 Objetivos Específicos

1. **Diseñar arquitectura modular: Cumplido.** El código se organiza en 15 módulos con responsabilidades claramente definidas, facilitando mantenimiento y extensión.
2. **Implementar 22 comandos: Cumplido.** Todos los comandos especificados fueron implementados y validados con tests.
3. **Desarrollar sistema de jobs: Cumplido.** Sistema completo con prioridades, persistencia y timeouts operacional.
4. **Evaluar desempeño bajo 3 perfiles: Cumplido.** Stress tests ejecutados con perfiles LOW, MEDIUM y HIGH, generando reportes detallados con métricas p50/p95/p99 y throughput.
5. **Análisis comparativo CPU-bound vs IO-bound: Cumplido.** Experimento dedicado comparó ambas categorías, identificando características de escalabilidad y cuellos de botella.
6. **Cobertura de tests $\geq$ 90 %: Cumplido.** Se alcanzó 90 % de cobertura mediante 146 tests unitarios.

7.3 Contribuciones

Las principales contribuciones de este trabajo son:

1. **Demostración práctica de conceptos de SO:** El proyecto materializa conceptos abstractos (concurrencia, sincronización, planificación) en un sistema funcional, facilitando su comprensión.

2. **Arquitectura escalable con pools especializados:** El diseño de tres pools independientes constituye un patrón aplicable a otros sistemas que requieren procesar cargas heterogéneas.

3. **Análisis empírico de trade-offs:** La evaluación cuantitativa de decisiones arquitectónicas (threads vs async, pools separados vs unificado) proporciona datos concretos para justificar elecciones de diseño.

4. **Código bien documentado y testeado:** El proyecto puede servir como referencia educativa para futuros estudiantes, mostrando buenas prácticas de ingeniería de software.

5. **Estudio de caso de Rust en sistemas concurrentes:** Demuestra cómo el ownership system de Rust previene clases enteras de bugs sin sacrificar desempeño.

7.4 Resultados Principales

Los experimentos de desempeño revelaron:

- **Latencias excelentes:** p50 de 8-11ms y p95 de 10-18ms bajo carga media, superando expectativas para un servidor educativo.
- **Throughput sostenido:** 250 req/s con 100 clientes concurrentes, apropiado para APIs internas de tamaño medio.
- **Escalabilidad sublineal:** Doubling workers no duplica throughput debido a la Ley de Amdahl, con saturación alrededor de 8-12 workers.
- **Pools separados beneficiosos:** Comandos CPU-bound no bloquean comandos básicos, validando la decisión arquitectónica.
- **Backpressure efectivo:** El sistema se mantiene estable bajo sobrecarga, rechazando requests gracefully.

7.5 Limitaciones

Es importante reconocer las limitaciones del sistema:

- **Throughput modesto:** 250 req/s es bajo comparado con servidores de producción (Nginx: 100k+ req/s), aunque apropiado para el contexto educativo.

- **Solo HTTP/1.0:** Carece de features modernas como keep-alive (HTTP/1.1) o multiplexing (HTTP/2).
- **Sin TLS/SSL:** Comunicaciones en texto plano, no apto para producción en internet.
- **Persistencia simple:** JSON plano sin ACID ni optimizaciones, no escalable a millones de jobs.
- **Lock contention:** Bajo carga extrema, el Mutex compartido se convierte en cuello de botella.

Estas limitaciones son aceptables dado el objetivo educativo del proyecto, pero deberían ser abordadas en una implementación de producción.

7.6 Lecciones Aprendidas

7.6.1 Técnicas

- El ownership system de Rust es altamente efectivo para prevenir bugs de concurrencia, aunque requiere inversión en la curva de aprendizaje.
- Tests automáticos con alta cobertura son esenciales para mantener la calidad del código a medida que evoluciona.
- Instrumentación con métricas desde el inicio facilita identificar cuellos de botella sin necesidad de re-instrumentar después.

7.6.2 Conceptuales

- La concurrencia introduce complejidad no trivial. Incluso con las garantías de Rust, razonar sobre comportamiento concurrente requiere cuidado.
- Toda decisión arquitectónica tiene trade-offs. El contexto y los objetivos determinan qué trade-off es aceptable.
- La Ley de Amdahl es un límite fundamental. Ninguna cantidad de paralelización puede superar completamente las partes secuenciales.
- Medir es más confiable que intuir. El profiling revela cuellos de botella inesperados.

7.6.3 Prácticas

- El diseño modular facilita agregar funcionalidades sin grandes refactorizaciones.
- La documentación clara ahorra tiempo al retomar el proyecto después de pausas.
- Control de versiones con commits atómicos y mensajes descriptivos es invaluable.

7.7 Trabajo Futuro

Este proyecto abre múltiples líneas de investigación y desarrollo futuro:

7.7.1 Investigación

- **Comparación threads vs async/await:** Implementar versión equivalente con Tokio y comparar métricamente ambos enfoques bajo cargas idénticas.
- **Estudio de fairness:** Analizar si los jobs de baja prioridad sufren starvation bajo carga alta de jobs urgentes.
- **Análisis de memory layout:** Estudiar cómo diferentes estructuras de datos (VecDeque, LinkedList, ArrayQueue) afectan cache locality y desempeño.
- **Impacto de NUMA:** En sistemas multi-socket, evaluar cómo la afinidad de threads a cores afecta latencias.

7.7.2 Desarrollo

- **Implementación HTTP/2:** Agregar soporte para multiplexing, server push y header compression.
- **Modo cluster:** Extender el servidor para correr en múltiples nodos con balanceo de carga distribuido.
- **Caché distribuido:** Integrar Redis para cachear resultados de comandos costosos.
- **Observabilidad avanzada:** Exportar métricas a Prometheus y agregar distributed tracing con OpenTelemetry.

7.8 Reflexión Final

Este proyecto demostró que es posible construir un servidor HTTP concurrente funcional desde cero, aplicando rigurosamente conceptos de sistemas operativos. Si bien el desempeño no alcanza niveles de producción, el ejercicio proporcionó comprensión profunda de los desafíos inherentes a sistemas concurrentes y las estrategias para abordarlos.

La elección de Rust como lenguaje de implementación resultó acertada. Su sistema de tipos previno numerosos bugs que en lenguajes como C habrían requerido horas de debugging con herramientas como Valgrind o AddressSanitizer. El costo fue una curva de aprendizaje más empinada, pero el beneficio en seguridad y confiabilidad compensó la inversión.

El enfoque basado en pools de workers especializados demostró ser una estrategia efectiva para balancear simplicidad con desempeño razonable. Aunque existen arquitecturas más escalables (event-driven, actor model), la claridad conceptual de este diseño lo hace apropiado para contextos educativos.

Finalmente, este proyecto reforzó la importancia de medir antes de optimizar. Varias suposiciones iniciales sobre dónde estarían los cuellos de botella resultaron incorrectas. Solo mediante profiling y benchmarking sistemático fue posible identificar las partes del código que realmente impactaban el desempeño.

En conclusión, el servidor HTTP implementado cumple satisfactoriamente con los objetivos establecidos, demostrando tanto la viabilidad técnica del diseño como la efectividad de las herramientas y prácticas empleadas.

Referencias

[1] A. S. Tanenbaum and H. Bos, *Modern Operating Systems*, 4th ed. Pearson, 2015.

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

[3] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*, 2nd ed. Morgan Kaufmann, 2020.

[4] T. Berners-Lee, R. Fielding, and H. Frystyk, "RFC 1945: Hypertext transfer protocol – HTTP/1.0," Internet Engineering Task Force (IETF), May 1996. [Online]. Available: https://www.rfc-editor.org/rfc/rfc1945

[5] The Rust Team, "The rust programming language," https://doc.rust-lang.org/book/, 2023, accessed: 2024-10-30.

[6] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *AFIPS Spring Joint Computer Conference*, Atlantic City, NJ, 1967, pp. 483–485.

A Configuración del Sistema

```
Configuración del servidor:
- workers-cpu: 4
- workers-io: 4
- workers-basic: 2
- queue-cpu: 64
- queue-io: 64
- queue-basic: 32
- timeout-cpu: 60000ms
- timeout-io: 60000ms
- timeout-basic: 30000ms
```

B Comandos Implementados

B.1 Comandos Básicos

1. `/status` - Estado del servidor
2. `/fibonacci?num=N` - Cálculo de Fibonacci
3. `/reverse?text=STRING` - Inversión de texto
4. `/toupper?text=STRING` - Conversión a mayúsculas
5. `/timestamp` - Timestamp actual
6. `/random?min=N&max=M` - Número aleatorio
7. `/hash?text=STRING&algo=sha256` - Hash SHA256
8. `/createfile?name=FILE&content=TEXT&repeat=N` - Crear archivo
9. `/deletefile?name=FILE` - Eliminar archivo
10. `/simulate?ms=N` - Simular carga
11. `/sleep?ms=N` - Dormir N milisegundos
12. `/loadtest?requests=N` - Test de carga
13. `/help` - Ayuda

B.2 Comandos CPU-bound

1. `/isprime?num=N` - Test de primalidad
2. `/factor?num=N` - Factorización
3. `/pi?iterations=N` - Cálculo de Pi (Monte Carlo)
4. `/mandelbrot?width=W&height=H&max_iter=I` - Conjunto de Mandelbrot
5. `/matrixmul?size=N` - Multiplicación de matrices

B.3 Comandos IO-bound

1. `/sortfile?name=FILE&algo=merge|quick` - Ordenar archivo

2. `/wordcount?name=FILE` - Contar palabras
3. `/grep?name=FILE&pattern=REGEX` - Búsqueda con regex
4. `/compress?name=FILE&codec=gzip` - Compresión
5. `/hashfile?name=FILE&algo=sha256` - Hash de archivo