

PRINCIPIOS DE SISTEMAS OPERATIVOS

Informe Técnico Extendido

Profesor: Kenneth Obando Rodríguez

Estudiantes:

Erick Kauffmann Porcar, CI: 2022180244

Jose Sequeira, CI: 2019380131

Luany Masís Aagesen, CI: 2022155917

25 de junio de 2025

Resumen

Este informe describe el diseño, implementación y evaluación de un sistema distribuido desarrollado en Rust, enfocado en el procesamiento paralelo de tareas como conteo de palabras y multiplicación de matrices. Se analizan la arquitectura, los protocolos de comunicación, la tolerancia a fallos, la escalabilidad y se presentan resultados comparativos con la ejecución local.

Índice

1. Arquitectura del sistema distribuido	2
1.1. Inicialización y roles del sistema	2
1.2. Modelo cliente-servidor y concurrencia	2
1.3. Ruteo y procesamiento de solicitudes	2
1.4. División y procesamiento distribuido de tareas	2
1.5. Almacenamiento distribuido y persistencia	3
1.6. Gestión de errores y estado	3
1.7. Extensibilidad y mantenibilidad	3
1.8. Despliegue e integración con Kubernetes (k8s)	3
2. Protocolos de comunicación	4
2.1. Protocolo de aplicación: HTTP simplificado sobre TCP	4
2.2. Manejo de solicitudes y respuestas	4
2.3. Comunicación con Redis	5
2.4. Seguridad y validación	5
3. Análisis de tolerancia a fallos y escalabilidad	5
3.1. Tolerancia a fallos	5
3.2. Escalabilidad	6
4. Resultados	6
4.1. Resultados de pruebas unitarias	6
4.2. Análisis de cobertura de código	6
4.3. Cobertura visual por función	7
5. Conclusiones	7

1. Arquitectura del sistema distribuido

El sistema está diseñado bajo una arquitectura modular y escalable, orientada a la ejecución distribuida de tareas computacionales intensivas, como el conteo de palabras en archivos de texto y la multiplicación de matrices. A continuación, se detallan los principales componentes y su interacción:

1.1. Inicialización y roles del sistema

El punto de entrada (`main.rs`) permite que la aplicación se ejecute en diferentes roles, determinados por la variable de entorno `APP_ROLE`:

- **Master:** Encargado de coordinar la distribución de tareas y, opcionalmente, de interactuar con un sistema de almacenamiento distribuido como Redis.
- **Slave:** Recibe instrucciones del master y ejecuta tareas específicas.
- **Standalone:** Permite la ejecución local para pruebas y desarrollo.

Esta flexibilidad facilita tanto la ejecución distribuida como la local, permitiendo comparar el rendimiento y la escalabilidad.

1.2. Modelo cliente-servidor y concurrencia

El servidor principal (`server.rs`) escucha conexiones TCP en un puerto configurable. Utiliza un pool de hilos (implementado en `pool/threadpool.rs`) para manejar múltiples solicitudes concurrentes. Cada hilo (`worker`) toma tareas de una cola interna (canal de mensajes) y las ejecuta de forma independiente, lo que permite aprovechar los recursos de hardware y mejorar la capacidad de respuesta bajo carga.

El pool de hilos se construye con un tamaño fijo, y cada `worker` ejecuta un bucle donde espera nuevas tareas, las procesa y actualiza su estado. El diseño asegura que los trabajos en curso se completen antes de cerrar el sistema, proporcionando un apagado controlado.

1.3. Ruteo y procesamiento de solicitudes

El módulo de rutas (`server/routes.rs`) actúa como el despachador central de solicitudes. Cada endpoint corresponde a una funcionalidad específica (por ejemplo, `countpartial`, `matrixtotal`, `fibonacci`, etc.). El ruteo se basa en el análisis de la URI y los parámetros de la solicitud, permitiendo una extensión sencilla para nuevas operaciones.

Las solicitudes pueden ser de distintos tipos (GET, POST, DELETE), y el sistema responde con códigos estándar y mensajes claros en caso de error o éxito.

1.4. División y procesamiento distribuido de tareas

Las tareas computacionales se dividen en partes para su procesamiento paralelo:

- **Conteo de palabras:** El archivo de texto se divide en fragmentos según el número de partes especificado. Cada fragmento es procesado por un `worker`, que cuenta las palabras en su sección (`distributed/count_partial.rs`). Los resultados parciales se combinan posteriormente para obtener el total.

- **Multiplicación de matrices:** Cada celda de la matriz resultado puede ser calculada de forma independiente (`distributed/matrix_partial.rs`). Los workers calculan celdas específicas y luego se ensamblan los resultados.

Este enfoque permite escalar horizontalmente el procesamiento, ya que cada worker puede ejecutarse en diferentes nodos o hilos.

1.5. Almacenamiento distribuido y persistencia

El sistema puede integrarse con Redis (`redis_comm/connection.rs`) para almacenar resultados parciales o coordinar el estado entre nodos distribuidos. Esto es especialmente útil en escenarios donde los workers se ejecutan en diferentes máquinas y se requiere persistencia o intercambio rápido de datos.

1.6. Gestión de errores y estado

El sistema implementa módulos específicos para el manejo de errores (`errors/`) y el seguimiento del estado de los workers y del servidor (`status/`). Esto permite detectar y registrar fallos, así como monitorear la salud y el rendimiento del sistema.

1.7. Extensibilidad y mantenibilidad

La arquitectura modular, basada en la separación clara de responsabilidades, facilita la incorporación de nuevas funcionalidades, la corrección de errores y la adaptación a diferentes escenarios de despliegue (local, distribuido, con o sin almacenamiento externo).

1.8. Despliegue e integración con Kubernetes (k8s)

Para facilitar la orquestación, escalabilidad y gestión de los diferentes componentes del sistema distribuido, se provee un archivo de despliegue para Kubernetes: `k8s-deployment.yaml`.

Este archivo define los siguientes recursos principales:

- **Redis:** Se despliega como un contenedor independiente, accesible mediante un servicio interno, para el almacenamiento y coordinación de datos entre los nodos.
- **Master:** Un deployment con un único pod que ejecuta la aplicación en modo maestro, expuesto mediante un servicio de tipo `LoadBalancer` para recibir solicitudes externas.
- **Slaves:** Un deployment con múltiples réplicas (por defecto 3), cada una ejecutando la aplicación en modo esclavo. Los slaves se comunican con el master y con Redis a través de los servicios internos del clúster.

La configuración de variables de entorno en cada contenedor (`APP_ROLE`, `PORT`, `REDIS_URL`, `MASTER_URL`) permite que cada pod adopte el rol adecuado y se conecte correctamente a los servicios necesarios. Esto facilita la escalabilidad horizontal (aumentando el número de slaves) y la tolerancia a fallos, ya que Kubernetes puede reiniciar pods caídos automáticamente.

El uso de Kubernetes permite desplegar el sistema en entornos de nube o en clústeres locales, gestionando de forma eficiente los recursos y la comunicación entre los diferentes componentes distribuidos.

2. Protocolos de comunicación

El sistema implementa varios niveles de protocolos de comunicación para coordinar la interacción entre clientes, servidores y servicios auxiliares:

2.1. Protocolo de aplicación: HTTP simplificado sobre TCP

La comunicación principal entre clientes y el servidor se realiza sobre el protocolo TCP, utilizando un protocolo de aplicación inspirado en HTTP/1.1. Cada mensaje de solicitud incluye:

- **Línea de petición:** Método (GET, POST, DELETE), URI, versión y parámetros de consulta.
- **Cabeceras:** Información adicional como Content-Type y Content-Length.
- **Cuerpo:** Datos en formato JSON o x-www-form-urlencoded, según la operación.

El servidor parsea cada solicitud utilizando un analizador propio (`parser.rs`), que valida la sintaxis, extrae los parámetros y construye una estructura interna (`HttpRequest`). Los endpoints se enrutan según la URI y el método, y se responde con una estructura `HttpResponse` que incluye:

- **Código de estado:** 200 (OK), 400 (Bad Request), 404 (Not Found), 405 (Method Not Allowed), 501 (Not Implemented), 505 (HTTP Version Not Supported), 507 (Insufficient Storage), 500 (Internal Server Error).
- **Razón:** Descripción textual del estado.
- **Cabeceras y cuerpo:** Según corresponda.

2.2. Manejo de solicitudes y respuestas

El servidor utiliza un modelo asíncrono basado en hilos para procesar múltiples solicitudes concurrentes. Cada conexión TCP es gestionada por un worker del pool de hilos, que se encarga de:

1. Leer y parsear la solicitud entrante.
2. Validar el método, la versión y los parámetros.
3. Ejecutar la operación solicitada (por ejemplo, conteo parcial, multiplicación de matrices, manipulación de archivos).
4. Construir y enviar la respuesta adecuada al cliente.

El diseño modular del parser y las estructuras de solicitud/respuesta permite extender fácilmente el protocolo para nuevas operaciones o tipos de datos.

2.3. Comunicación con Redis

Para la coordinación y almacenamiento distribuido, el sistema utiliza Redis como base de datos en memoria. La comunicación con Redis se realiza mediante el protocolo nativo de Redis, utilizando la biblioteca oficial en Rust. Los workers pueden conectarse a Redis para almacenar resultados parciales, recuperar datos o sincronizar el estado entre nodos distribuidos.

La URL de conexión a Redis se configura mediante variables de entorno y puede apuntar a un servicio interno del clúster (por ejemplo, `redis://redis-service:6379` en despliegues de Kubernetes).

2.4. Seguridad y validación

El sistema valida cuidadosamente la sintaxis de las solicitudes, los tipos de datos y los parámetros, devolviendo códigos de error apropiados ante cualquier inconsistencia. Esto previene ataques comunes como inyección de comandos o desbordamientos de búfer, y garantiza la robustez de la comunicación.

3. Análisis de tolerancia a fallos y escalabilidad

El sistema ha sido diseñado considerando tanto la tolerancia a fallos como la escalabilidad, aspectos fundamentales en entornos distribuidos y de alta concurrencia.

3.1. Tolerancia a fallos

- **Manejo de errores en el servidor:** El servidor implementa un manejo robusto de errores en cada etapa del procesamiento de solicitudes. Si ocurre un error irreparable (por ejemplo, fallo al iniciar el servidor o el pool de hilos), el sistema lo registra y termina de forma controlada, evitando estados inconsistentes.
- **Aislamiento de fallos por hilos:** Cada solicitud es gestionada por un hilo independiente dentro del pool. Si un hilo encuentra un error o entra en pánico, el resto del sistema sigue funcionando, y el hilo puede ser reiniciado por el pool.
- **Persistencia y recuperación con Redis:** Al utilizar Redis como almacenamiento intermedio, los resultados parciales pueden ser recuperados en caso de reinicio de algún nodo, evitando la pérdida de progreso en tareas distribuidas.
- **Orquestación con Kubernetes:** Kubernetes monitoriza el estado de los pods (master, slaves y Redis) y reinicia automáticamente cualquier pod que falle, asegurando alta disponibilidad y recuperación automática ante fallos de hardware o software.
- **Validación y control de parámetros:** El sistema valida cuidadosamente los parámetros de entrada y la integridad de los datos, previniendo errores lógicos y ataques por entradas maliciosas.

3.2. Escalabilidad

- **Escalabilidad horizontal:** Gracias a la arquitectura basada en Kubernetes, es posible aumentar el número de instancias de slaves simplemente ajustando el parámetro de réplicas en el deployment. Esto permite procesar más tareas en paralelo y adaptarse dinámicamente a la carga de trabajo.
- **Escalabilidad vertical:** El pool de hilos en cada instancia puede ser configurado para aprovechar mejor los recursos de hardware disponibles (CPU y memoria), incrementando la capacidad de procesamiento de cada pod.
- **Balanceo de carga:** El servicio de tipo LoadBalancer en Kubernetes distribuye las solicitudes entrantes entre los pods disponibles, evitando cuellos de botella y mejorando la eficiencia global del sistema.
- **Desacoplamiento de componentes:** La separación entre master, slaves y Redis permite escalar cada componente de forma independiente según las necesidades del sistema.
- **Limitaciones:** Aunque el sistema es escalable, existen límites prácticos impuestos por el tamaño del pool de hilos, la capacidad de la red y la latencia de comunicación entre nodos. El diseño modular facilita la identificación y mejora de estos cuellos de botella.

En conjunto, estos mecanismos permiten que el sistema mantenga un alto nivel de disponibilidad y rendimiento, incluso ante fallos parciales o incrementos significativos en la demanda.

4. Resultados

4.1. Resultados de pruebas unitarias

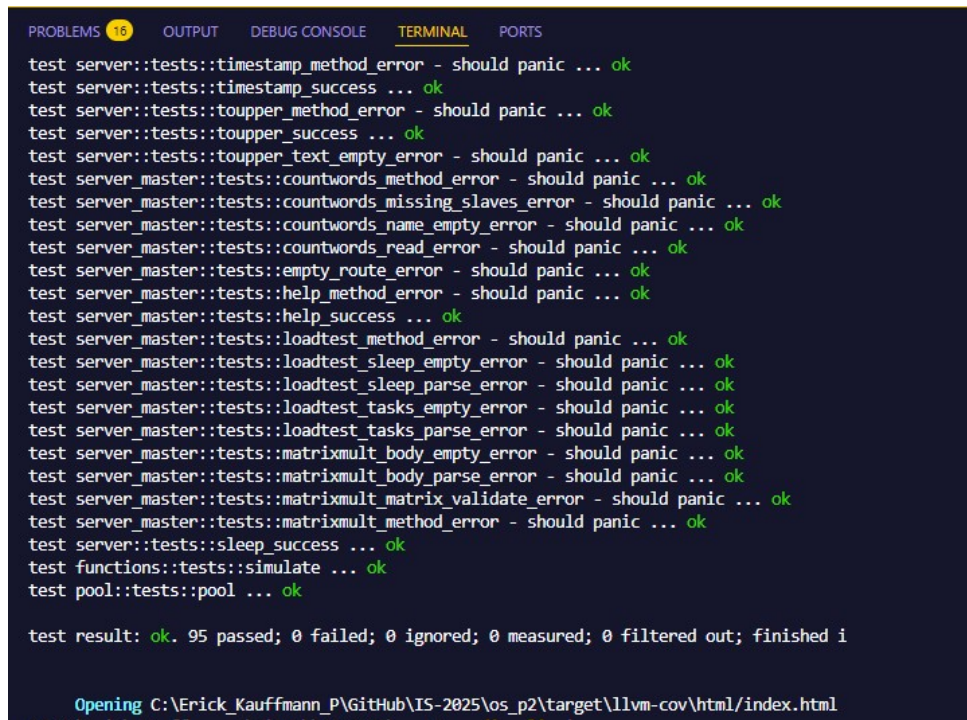
Se realizaron pruebas unitarias utilizando `cargo test`, obteniendo un resultado satisfactorio con un total de **95 pruebas ejecutadas** y **0 fallos**. Las pruebas cubren tanto casos exitosos como rutas de error, validando el comportamiento esperado ante entradas vacías, métodos no soportados, parámetros inválidos y operaciones correctas.

4.2. Análisis de cobertura de código

Se utilizó la herramienta `cargo llvm-cov` para generar un reporte de cobertura de código. Los resultados muestran una cobertura general del **66.60 % de líneas** y **66.55 % de funciones**, lo cual es adecuado considerando la extensión del sistema y la presencia de múltiples rutas de código para manejo de errores.

Los módulos con mayor cobertura son:

- `functions/*`, `distributed/*` y `pool/*`: con cobertura del 100 %.
- `server/routes.rs`: 94.74 % de funciones y 90.03 % de líneas.
- `server/mod.rs` y `server_master/mod.rs`: cobertura casi total.



```
PROBLEMS 16 OUTPUT DEBUG CONSOLE TERMINAL PORTS
test server::tests::timestamp_method_error - should panic ... ok
test server::tests::timestamp_success ... ok
test server::tests::toupper_method_error - should panic ... ok
test server::tests::toupper_success ... ok
test server::tests::toupper_text_empty_error - should panic ... ok
test server_master::tests::countwords_method_error - should panic ... ok
test server_master::tests::countwords_missing_slaves_error - should panic ... ok
test server_master::tests::countwords_name_empty_error - should panic ... ok
test server_master::tests::countwords_read_error - should panic ... ok
test server_master::tests::empty_route_error - should panic ... ok
test server_master::tests::help_method_error - should panic ... ok
test server_master::tests::help_success ... ok
test server_master::tests::loadtest_method_error - should panic ... ok
test server_master::tests::loadtest_sleep_empty_error - should panic ... ok
test server_master::tests::loadtest_sleep_parse_error - should panic ... ok
test server_master::tests::loadtest_tasks_empty_error - should panic ... ok
test server_master::tests::loadtest_tasks_parse_error - should panic ... ok
test server_master::tests::matrixmult_body_empty_error - should panic ... ok
test server_master::tests::matrixmult_body_parse_error - should panic ... ok
test server_master::tests::matrixmult_matrix_validate_error - should panic ... ok
test server_master::tests::matrixmult_method_error - should panic ... ok
test server::tests::sleep_success ... ok
test functions::tests::simulate ... ok
test pool::tests::pool ... ok

test result: ok. 95 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished i

Opening C:\Erick_Kauffmann_P\GitHub\IS-2025\os_p2\target\llvm-cov\html\index.html
```

Figura 1: Resultados de las pruebas unitarias con cargo test

Los módulos que aún presentan oportunidades de mejora incluyen:

- errors/* y redis_comm/*: sin cobertura actual.
- main.rs, status.rs, server/server.rs, server_master/routes.rs y otros módulos de conexión o entrada.

4.3. Cobertura visual por función

En el análisis visual se pueden identificar con claridad las líneas no cubiertas dentro de los módulos. Las líneas en rojo indican funciones que no han sido alcanzadas durante la ejecución de las pruebas unitarias.

5. Conclusiones

El sistema distribuido desarrollado permite mejorar el rendimiento en tareas paralelizables mediante el uso de hilos y comunicación TCP. La arquitectura modular facilita la extensión y el mantenimiento. Se identifican oportunidades de mejora en la escalabilidad y la tolerancia a fallos ante escenarios extremos.

Referencias

- [1] Rust Programming Language. <https://www.rust-lang.org/>
- [2] TCP/IP Protocol. <https://datatracker.ietf.org/doc/html/rfc793>

Filename	Function Coverage	Line Coverage	Region Coverage	Branch Coverage
<code>distributed\count_partial.rs</code>	100.00% (3/3)	100.00% (28/28)	100.00% (14/14)	- (0/0)
<code>distributed\count_total.rs</code>	100.00% (1/1)	100.00% (3/3)	100.00% (1/1)	- (0/0)
<code>distributed\matrix_partial.rs</code>	100.00% (1/1)	100.00% (9/9)	100.00% (5/5)	- (0/0)
<code>distributed\matrix_total.rs</code>	100.00% (2/2)	83.33% (20/24)	90.00% (18/20)	- (0/0)
<code>distributed\mod.rs</code>	100.00% (8/8)	100.00% (66/66)	100.00% (19/19)	- (0/0)
<code>errors\implement.rs</code>	0.00% (0/1)	0.00% (0/3)	0.00% (0/1)	- (0/0)
<code>errors\matrix.rs</code>	100.00% (2/2)	100.00% (6/6)	100.00% (2/2)	- (0/0)
<code>errors\mod.rs</code>	0.00% (0/4)	0.00% (0/30)	0.00% (0/4)	- (0/0)
<code>errors\parse.rs</code>	0.00% (0/1)	0.00% (0/3)	0.00% (0/1)	- (0/0)
<code>errors\pool.rs</code>	0.00% (0/1)	0.00% (0/3)	0.00% (0/1)	- (0/0)
<code>errors\server.rs</code>	0.00% (0/1)	0.00% (0/3)	0.00% (0/1)	- (0/0)
<code>errors\slaves.rs</code>	0.00% (0/2)	0.00% (0/6)	0.00% (0/2)	- (0/0)
<code>functions\createfile.rs</code>	100.00% (1/1)	100.00% (9/9)	88.89% (8/9)	- (0/0)
<code>functions\deletefile.rs</code>	100.00% (1/1)	100.00% (3/3)	100.00% (1/1)	- (0/0)
<code>functions\fibonacci.rs</code>	100.00% (1/1)	100.00% (14/14)	100.00% (10/10)	- (0/0)
<code>functions\hash.rs</code>	100.00% (1/1)	100.00% (3/3)	100.00% (1/1)	- (0/0)
<code>functions\help.rs</code>	100.00% (1/1)	100.00% (20/20)	100.00% (1/1)	- (0/0)
<code>functions\mod.rs</code>	100.00% (16/16)	93.10% (108/116)	84.44% (38/45)	- (0/0)
<code>functions\random.rs</code>	100.00% (2/2)	100.00% (8/8)	100.00% (6/6)	- (0/0)
<code>functions\reverse.rs</code>	100.00% (1/1)	100.00% (3/3)	100.00% (1/1)	- (0/0)
<code>functions\simulate.rs</code>	100.00% (1/1)	100.00% (4/4)	100.00% (1/1)	- (0/0)
<code>functions\sleep.rs</code>	100.00% (1/1)	100.00% (4/4)	100.00% (1/1)	- (0/0)
<code>functions\timestamp.rs</code>	100.00% (1/1)	100.00% (4/4)	100.00% (1/1)	- (0/0)
<code>functions\toupper.rs</code>	100.00% (1/1)	100.00% (3/3)	100.00% (1/1)	- (0/0)
<code>main.rs</code>	0.00% (0/1)	0.00% (0/20)	0.00% (0/12)	- (0/0)

Figura 2: Cobertura de funciones y líneas de código (parte 1)

<code>models\matrix.rs</code>	100.00% (1/1)	82.35% (14/17)	80.00% (12/15)	- (0/0)
<code>models\request.rs</code>	66.67% (2/3)	21.67% (13/60)	8.00% (2/25)	- (0/0)
<code>models\response.rs</code>	60.00% (3/5)	28.95% (22/76)	35.71% (10/28)	- (0/0)
<code>models\status.rs</code>	50.00% (3/6)	37.10% (23/62)	25.00% (6/24)	- (0/0)
<code>pool\mod.rs</code>	100.00% (3/3)	100.00% (17/17)	100.00% (7/7)	- (0/0)
<code>pool\threadpool.rs</code>	100.00% (5/5)	100.00% (60/60)	92.59% (25/27)	- (0/0)
<code>redis_comm\connection.rs</code>	0.00% (0/1)	0.00% (0/4)	0.00% (0/6)	- (0/0)
<code>redis_comm\count_store.rs</code>	0.00% (0/3)	0.00% (0/24)	0.00% (0/26)	- (0/0)
<code>redis_comm\matrix_store.rs</code>	0.00% (0/14)	0.00% (0/94)	0.00% (0/52)	- (0/0)
<code>server\mod.rs</code>	100.00% (53/53)	95.60% (847/886)	63.21% (67/106)	- (0/0)
<code>server\parser.rs</code>	0.00% (0/5)	0.00% (0/118)	0.00% (0/66)	- (0/0)
<code>server\routes.rs</code>	94.74% (18/19)	90.03% (307/341)	91.18% (155/170)	- (0/0)
<code>server\server.rs</code>	0.00% (0/5)	0.00% (0/89)	0.00% (0/48)	- (0/0)
<code>server_master\mod.rs</code>	100.00% (35/35)	99.30% (566/570)	98.90% (90/91)	- (0/0)
<code>server_master\parser.rs</code>	0.00% (0/9)	0.00% (0/123)	0.00% (0/72)	- (0/0)
<code>server_master\routes.rs</code>	35.29% (12/34)	27.72% (102/368)	31.09% (74/238)	- (0/0)
<code>server_master\server.rs</code>	0.00% (0/10)	0.00% (0/75)	0.00% (0/49)	- (0/0)
<code>server_master\slaves.rs</code>	37.50% (3/8)	15.00% (9/60)	18.75% (3/16)	- (0/0)
<code>status\status.rs</code>	66.67% (4/6)	52.94% (18/34)	44.44% (4/9)	- (0/0)
Totals	66.55% (187/281)	66.60% (2313/3473)	47.25% (584/1236)	- (0/0)

Generated by llvm-cov -- llvm version 19.1.7-rust-1.86.0-stable

Figura 3: Cobertura de funciones y líneas de código (parte 2)

[3] Docker Official Rust Image. https://hub.docker.com/_/rust

[4] Docker Docs: Build images with Rust. <https://docs.docker.com/guides/rust/build-images/>