Introducción a Distribuidos

PROYECTO FINAL

Realizado por:

Juan Esteban Camargo

Diego Andres Martinez

Docente:

Rafael Paez Mendez

19/11/2025

Bogotá D.C, Colombia

1. Introducción y objetivo de los experimentos

En este documento se resumen los experimentos de rendimiento realizados sobre el sistema distribuido de préstamo de libros desarrollado en la asignatura de Introducción a Sistemas Distribuidos. El objetivo de las pruebas es caracterizar el comportamiento del sistema bajo diferentes niveles de concurrencia, midiendo latencia, throughput y tasa de fallos en las operaciones principales (PRESTAR, RENOVAR y DEVOLVER). Para ello se utilizó una arquitectura basada en ZeroMQ, con un gateway HTTP–ZMQ para integrar la herramienta de pruebas de carga Locust con los componentes reales del sistema.

A partir de estos experimentos se busca identificar cuellos de botella, validar la capacidad de escalamiento horizontal y evaluar el impacto de los patrones de comunicación REQ/REP y PUB/SUB bajo diferentes niveles de carga.

2. Especificaciones de hardware, software y herramientas

2.1 Hardware y despliegue

Máquina	Rol principal	SO / RAM	Componentes que ejecuta
Máquina 1 – PC Juano	Nodo de sede 1 y cliente	Windows 11, 16 GB RAM (o lo que tengas)	GC sede 1, actores de sede 1 (Préstamo, Renovación, Devolución) y proceso solicitante PS1
Máquina 2 – VM 10.43.102.221	Nodo de datos sede 1	Ubuntu 22.04, 8 GB RAM (ajusta)	Gestor de Almacenamiento GA1 y base de datos principal de la sede 1
Máquina 3 – VM 10.43.102.23	Nodo completo sede 2	Ubuntu 22.04, 8 GB RAM (ajusta)	Gestor de carga sede 2, actores sede 2, GA2 y base de datos réplica sede 2

La arquitectura de pruebas se compone de tres máquinas físicas/virtuales conectadas en la misma red. La sede 1 se ejecuta parcialmente en el PC local (gestor de carga y actores) y parcialmente en una VM con el Gestor de Almacenamiento y la base de datos principal. La sede 2 se ejecuta completamente en una segunda VM, que aloja su Gestor de Carga, actores, Gestor de Almacenamiento GA2 y la base de datos réplica. Entre GA1 y GA2 se mantiene una replicación asíncrona de los datos de préstamo.

2.2 Software

- **Lenguaje y runtime:** Python 3.x para los procesos distribuídos y el gateway HTTP–ZMQ.

- **Mensajería:** ZeroMQ usando patrones REQ/REP para la operación PRESTAR y PUB/SUB para RENOVAR y DEVOLVER.
- **Base de datos:** SQLite en modo WAL para la persistencia de préstamos y usuarios (si lo mencionas en el informe largo).
- **Gateway:** Servicio HTTP implementado con Flask que traduce las peticiones HTTP de Locust a mensajes ZeroMQ y devuelve las respuestas como JSON.

2.3 Herramientas de medición

Para la generación de carga y la obtención de métricas de rendimiento se utilizó Locust, framework de pruebas de carga que permite definir escenarios de usuarios concurrentes y registrar automáticamente métricas como latencia por operación, requests por segundo (RPS), distribución de tiempos de respuesta y tasa de éxito/error.

Adicionalmente se analizaron los logs del gateway HTTP–ZMQ y de los procesos backend para correlacionar fallos y timeouts observados en Locust con el comportamiento interno del sistema.

3. Metodología experimental

Se definió un único escenario de pruebas (Experimento II), en el cual se mantiene constante la topología de tres máquinas (GC/Actores, GA1–GA2 y cliente PS/Locust) y se varían los usuarios concurrentes generados por Locust. La replicación de datos se configura de forma asíncrona desde GA1 hacia GA2 para emular dos sedes con consistencia eventual.

Parámetro	Valor
Usuarios concurrentes	4, 6 y 10 usuarios simulados en Locust
Duración por experimento	120 s
Spawn rate	2 usuarios/s
Operaciones	PRESTAR (REQ/REP), RENOVAR (PUB/SUB), DEVOLVER (PUB/SUB)

En cada ejecución, Locust envía solicitudes HTTP al gateway, que las traduce a objetos SolicitudOperacion y las reenvía al Gestor de Carga por REQ/REP. Desde allí, las operaciones de préstamo se procesan de forma sincrónica y las de renovación/devolución se propagan por los canales PUB/SUB hacia los actores correspondientes.

4. Resultados en tablas y gráficos

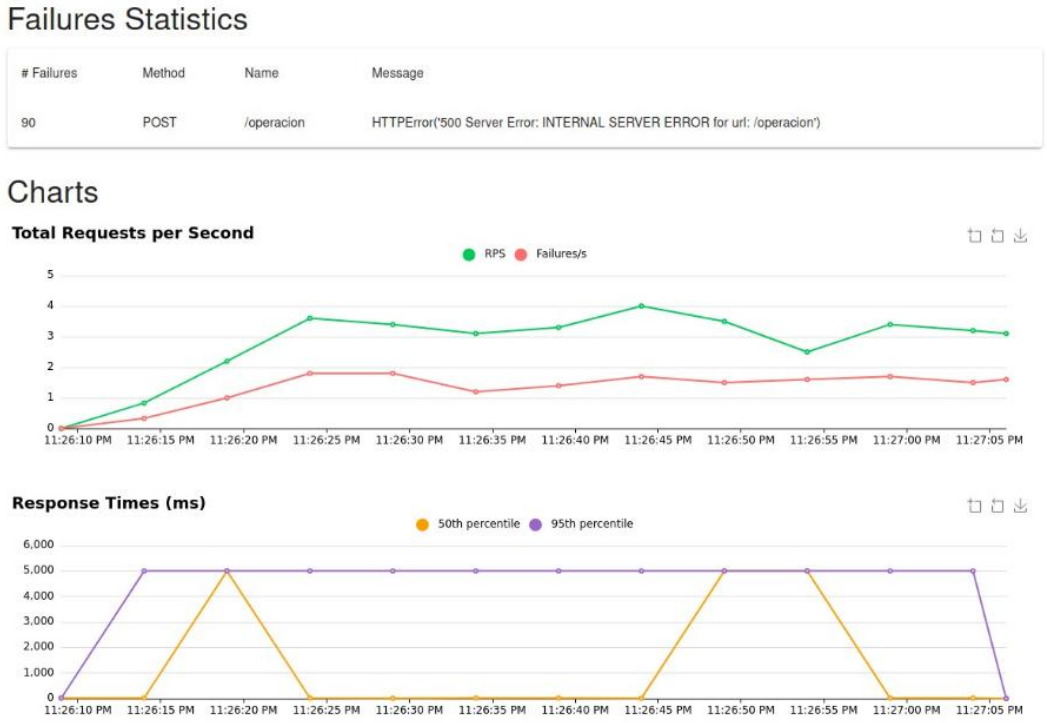
4.1 Resumen numérico

Usuarios	RPS promedio	Requests totales	Fallos	Latencia p50	Latencia p95	Error principal
4	1.54	89	35	13 ms	5000 ms	HTTP 500 – timeout REQ/REP

6	2.01	53	24	14 ms	5000 ms	HTTP 500 – timeout REQ/REP
10	3.24	193	90	14 ms	5000 ms	HTTP 500 – congestión REQ/REP

La tabla muestra que el throughput (RPS) crece casi linealmente con el número de usuarios, lo que evidencia capacidad de escalamiento horizontal del sistema. Sin embargo, la latencia típica (p50) se mantiene muy baja (13–14 ms) mientras que el p95 está anclado en 5000 ms, correspondiente al timeout configurado en el gateway HTTP–ZMQ, lo cual revela un cuello de botella en el patrón REQ/REP.

4.2 Gráficos



(Figura 1. Throughput, fallos y percentiles de tiempo de respuesta en el escenario de 10 usuarios.)

La parte superior muestra el total de solicitudes por segundo (RPS) y los fallos por segundo registrados por Locust: el RPS aumenta con la carga, mientras que los fallos aparecen cuando el canal REQ/REP se satura y el gateway alcanza el timeout configurado. La parte inferior presenta los percentiles de tiempo de respuesta (p50 y p95): el p50 confirma respuestas rápidas en condiciones normales, mientras que el p95 se mantiene cercano a 5 s, evidenciando el comportamiento bloqueante del patrón REQ/REP bajo alta concurrencia.

5. Análisis y conclusiones

Los resultados obtenidos muestran que el sistema distribuido de préstamo de libros es capaz de **escalar con el número de usuarios concurrentes**: a medida que se pasa de 4 a 10 usuarios, el throughput (RPS) aumenta de forma casi proporcional, sin que se observe una degradación inmediata en el comportamiento de los actores ni del Gestor de Almacenamiento. Esto indica que la separación en procesos solicitantes, gestor de carga, actores y gestor de almacenamiento permite aprovechar de forma adecuada los recursos de las máquinas involucradas.

Sin embargo, la brecha entre el percentil 50 y el percentil 95 revela un **cuello de botella importante en el patrón REQ/REP** utilizado por el gateway HTTP–ZMQ. Mientras el p50 se mantiene en el orden de milisegundos, el p95 coincide con el timeout configurado (≈ 5 s), lo que significa que una parte de las solicitudes queda bloqueada esperando respuesta y termina fallando por tiempo excedido. Estos timeouts se reflejan directamente como errores HTTP 500 en Locust y explican la presencia de fallos incluso cuando el RPS sigue creciendo.

En contraste, las operaciones RENOVAR y DEVOLVER, que se implementan mediante **PUB/SUB**, muestran un comportamiento más estable bajo carga, ya que no dependen de una respuesta sincrónica del Gestor de Carga. Esto sugiere que los **patrones asincrónicos** resultan más adecuados para operaciones que no requieren confirmación inmediata hacia el usuario, y que una futura iteración del sistema debería migrar parte de la lógica de préstamo hacia modelos no bloqueantes o basados en colas internas.

Otro resultado relevante es que la **replicación GA1→GA2 se mantuvo estable** durante los experimentos, garantizando consistencia eventual entre la base de datos principal y la réplica. En caso de fallo de GA1, la sede secundaria puede asumir el rol principal sin pérdida de información, lo que cumple con el objetivo de tolerancia a fallos planteado en el diseño del sistema.

En síntesis, las pruebas confirman que la arquitectura propuesta es viable y escalable, pero también evidencian que el gateway HTTP–ZMQ basado en REQ/REP se convierte en un **componente externo crítico** que concentra los tiempos de espera y condiciona la latencia p95. Como trabajo futuro se plantea reemplazar este componente por mecanismos completamente asincrónicos o por un bus de mensajería interno, con el fin de reducir la tasa de fallos y mejorar la experiencia percibida por los usuarios finales.