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

El presente documento corresponde a la segunda entrega del proyecto Sistema Distribuido de Préstamo de Libros, desarrollado en el marco de la asignatura *Sistemas Distribuidos*. En esta fase presentamos la implementación final del sistema, su funcionamiento real en múltiples máquinas, los mecanismos de replicación y tolerancia a fallos, y el análisis experimental del rendimiento bajo diferentes niveles de carga.

El objetivo del proyecto es diseñar una solución distribuida capaz de gestionar préstamos, devoluciones y renovaciones de libros entre dos sedes, garantizando disponibilidad, consistencia eventual y continuidad del servicio en presencia de fallos parciales. La arquitectura se desarrolló aplicando principios de modularidad, desacoplamiento y distribución horizontal, lo que permite escalar el sistema y mantener la operatividad aun cuando un componente falla.

Para la comunicación entre procesos se utilizó ZeroMQ, empleando los patrones REQ/REP para operaciones sincrónicas como el préstamo, y PUB/SUB para operaciones asincrónicas como devolución y renovación. Además, se implementó un canal adicional para replicación entre sedes y otro para heartbeat, que permite detectar la caída de la sede principal y activar el mecanismo de failover.

Cada sede cuenta con su propio Gestor de Carga (GC), Actores para cada tipo de operación, un Gestor de Almacenamiento (GA) y una base de datos SQLite. La replicación GA1→GA2 garantiza que la información se mantenga actualizada en ambas sedes, y que la sede secundaria pueda asumir la operación si la principal falla.

En esta entrega se presentan el modelo del sistema, los diagramas principales, las pruebas funcionales, las pruebas de rendimiento con 4, 6 y 10 procesos solicitantes, la validación del mecanismo de fallos y los resultados obtenidos. El documento integra diseño, desarrollo y experimentación para demostrar el correcto funcionamiento del sistema distribuido.

2. Modelos del Sistema

En esta sección presentamos las vistas esenciales del sistema distribuido desarrollado para gestionar préstamos, devoluciones y renovaciones de libros entre dos sedes. Los modelos permiten comprender cómo se organiza el sistema, cómo se comunican sus componentes y cómo se comporta ante diferentes escenarios de operación y fallos.

Cada modelo aquí descrito corresponde a decisiones técnicas que realmente aplicamos durante la implementación, por lo que existe coherencia directa entre lo documentado y el funcionamiento actual del proyecto.

2.1 Modelos del Sistema

El sistema se diseñó bajo una arquitectura distribuida conformada por dos sedes principales y una máquina cliente dedicada a la ejecución de los Procesos Solicitantes (PS). La organización general sigue un estilo cliente–servidor distribuido, con separación explícita de

responsabilidades, lo que facilita el escalamiento por sede, la tolerancia a fallos y la continuidad operacional ante caídas parciales.

Los elementos centrales que conforman el modelo arquitectural son los siguientes:

1. **Procesos Solicitantes (PS)**

Representan a los usuarios del sistema —profesores y estudiantes— que generan operaciones de préstamo, devolución o renovación. Cada PS lee un archivo de texto con múltiples solicitudes y las envía al Gestor de Carga correspondiente a su sede. Su función es actuar como punto de entrada al sistema, manteniendo un flujo constante de solicitudes.

2. **Gestor de Carga (GC)**

Opera como intermediario entre los Procesos Solicitantes y los Actores. Recibe las operaciones entrantes y determina su canal de procesamiento según el tipo de solicitud:

- Solicitudes de **préstamo**, manejadas de forma **síncrona (REQ/REP)**,
- Solicitudes de devolución y renovación, procesadas mediante comunicación asíncrona (PUB/SUB).

El GC es también responsable de reenviar solicitudes al actor apropiado, registrar eventos y participar en la detección de fallos mediante timeouts operativos.

3. **Actores**

Son procesos especializados encargados de ejecutar las operaciones sobre la base de datos. Cada uno cumple una función específica para desacoplar la carga del GC:

- **Actor de Préstamos:** procesa solicitudes de forma síncrona y valida la disponibilidad de ejemplares.
- **Actor de Devoluciones:** actualiza el estado de los libros mediante un canal asíncrono.
- **Actor de Renovaciones:** gestiona ampliaciones del periodo de préstamo sin bloquear operaciones entrantes.

4. **Gestor de Almacenamiento (GA)**

Es el componente que interactúa directamente con la base de datos. El sistema cuenta con dos instancias: una réplica primaria en la Sede 1 y una réplica secundaria en la Sede 2. Ambas mantienen la misma estructura y datos iniciales, sincronizándose de manera asíncrona para garantizar consistencia eventual entre sedes. El GA también administra el mecanismo de failover cuando la réplica primaria deja de estar disponible.

El modelo arquitectural contempla dos sedes físicas: Sede Norte y Sede Sur, cada una con su propio Gestor de Carga, su conjunto de Actores y su Gestor de Almacenamiento. Esta distribución permite que cada sede opere de forma autónoma, mantenga su propia base de datos local y participe en una replicación cruzada transparente para los usuarios.

Asimismo, se incluye un mecanismo de failover que garantiza continuidad del servicio en caso de caídas del GA primario. Este mecanismo utiliza un canal de heartbeat y la capacidad del GC para detectar fallos y redirigir solicitudes hacia la sede secundaria sin interrumpir el funcionamiento del sistema.

El modelo completo se representa en el diagrama de despliegue de la siguiente sección, en el que se visualiza la distribución física de los procesos en las máquinas utilizadas y la relación entre los componentes distribuidos.

2.1.1 Desglose de Componentes

- **Procesos Solicitantes:** Generan solicitudes a partir de archivos o simuladores de carga (JMETER). Cada Proceso Solicitante ejecuta peticiones válidas hacia el Gestor de Carga de su sede.
- **Gestor de Carga:** Atiende los requerimientos provenientes de los Procesos Solicitantes. Para las operaciones de devolución y renovación, responde inmediatamente y publica los datos en los tópicos correspondientes. Para las solicitudes de préstamo, espera confirmación del Actor antes de responder.
- **Actores:** Tres tipos de procesos especializados que escuchan los tópicos publicados por el Gestor de carga y actualizan el Gestor de almacenamiento. Cada uno se ejecuta de manera independiente para garantizar el paralelismo.
- **Gestor de Almacenamiento y Persistencia:** Gestiona las operaciones sobre la base de datos principal y actualiza la réplica de forma asíncrona. En caso de caída de la base principal, el sistema conmuta automáticamente hacia la réplica secundaria.

2.1.2 Tolerancia a Fallos

El sistema está diseñado con replicación activa–pasiva:

● El Gestor de Almacenamiento principal de la Sede 1 es la fuente de verdad.

● El Gestor de Almacenamiento de respaldo (Sede 2) se actualiza constantemente mediante colas de sincronización.

● Si el Gestor de Almacenamiento principal falla, la réplica toma el control mediante un proceso de health-check que monitorea la

disponibilidad del nodo principal.

2.2 Modelo de Interacción

2.2.2 Comunicación entre Componentes

La comunicación entre los procesos del sistema se apoya en el middleware ZeroMQ, que permite intercambiar mensajes de forma eficiente y flexible. En la implementación se utilizaron tres patrones fundamentales, cada uno seleccionado de acuerdo con el tipo de operación y el nivel de sincronización requerido.

En primer lugar, el patrón Request–Reply (REQ/REP) se emplea entre los Procesos Solicitantes (PS) y el Gestor de Carga (GC) para la operación de préstamo. En este caso se requiere una confirmación inmediata sobre la disponibilidad del libro, por lo que el PS envía la solicitud y bloquea su ejecución hasta recibir la respuesta. Este patrón asegura que la operación sea tratada de manera estrictamente sincrónica y facilita el manejo de errores explícitos.

Por otro lado, las operaciones de devolución y renovación se procesan mediante el patrón Publish–Subscribe (PUB/SUB). Aquí, el GC publica los eventos correspondientes y los actores suscritos los reciben de forma asíncrona. Este mecanismo permite atender múltiples operaciones simultáneamente sin bloquear la ejecución del GC, lo que mejora la capacidad de procesamiento en escenarios con alto número de solicitudes concurrentes.

Finalmente, para mantener la consistencia entre las bases de datos de ambas sedes, los Gestores de Almacenamiento (GA) utilizan un enlace asíncrono basado en Push–Pull (o un canal dedicado equivalentemente configurado) que transmite las actualizaciones realizadas en la sede principal hacia la sede réplica. Este canal funciona en segundo plano y permite que la sincronización sea continua sin afectar el desempeño de las operaciones atendidas en tiempo real. Gracias a este patrón se garantiza consistencia eventual y se habilita la conmutación automática en caso de fallo del GA principal.

2.2.3 Flujo de Interacción

El flujo general de interacción del sistema describe cómo una solicitud recorre los distintos componentes hasta completarse. Cada operación sigue una secuencia definida que garantiza orden, coherencia y correcta actualización de los datos entre sedes.

1. Solicitud desde los Procesos Solicitantes (PS)

El usuario genera una operación de préstamo, devolución o renovación. El Proceso Solicitante encapsula la información en un mensaje de tipo *SolicitudOperacion* y la envía al Gestor de Carga (GC) mediante ZeroMQ.

2. Procesamiento en el Gestor de Carga

Al recibir el mensaje, el GC identifica el tipo de operación y define el tratamiento adecuado:

- **Devolución o renovación:**
  Estas operaciones se gestionan de forma asíncrona. El GC envía una confirmación inmediata al PS indicando que la solicitud fue recibida y, posteriormente, publica el evento mediante el canal *PUB* para que los actores suscritos lo procesen.

- **Préstamo:**
  Este tipo de operación requiere validación sincrónica. El GC reenvía la solicitud al **Actor de Préstamo**, espera la respuesta oficial (libro disponible o no) y solo entonces devuelve un resultado definitivo al PS. Esto garantiza consistencia y evita conflictos en la disponibilidad del ejemplar.

3. Actores de negocio

Los actores suscritos reciben las solicitudes provenientes del GC y ejecutan la lógica correspondiente.
Para cada operación:

- Consultan y modifican el estado interno del recurso (libro o préstamo).

- Interactúan con el Gestor de Almacenamiento (GA) para ejecutar la operación en la base de datos.

- Generan una notificación interna o respuesta según lo requiera el flujo.

4. Persistencia y replicación

El **GA** escribe la actualización en la base de datos de la sede donde se originó la operación.
Una vez confirmada la operación, se activa el mecanismo de replicación:

- Cada cambio aplicado en la sede primaria se envía a la sede secundaria a través del canal de replicación.

- El GA de la réplica recibe los cambios y los aplica localmente, garantizando consistencia eventual entre ambas bases de datos.

Este proceso se realiza de manera asíncrona para no afectar la latencia percibida por los usuarios.

5. Respuesta final y registro en logs

El resultado de la operación se envía nuevamente al Proceso Solicitante.
En paralelo, el sistema registra en los logs:

- El tipo de operación

- La sede que procesó la solicitud
- El resultado (éxito o fallo)
- Timestamps y datos útiles para métricas e inspección posterior

Este registro permite reconstruir el flujo de cada operación y sirve como soporte para auditoría, análisis y detección de fallos.

2.3 Modelo de Fallos

En un sistema distribuido como el desarrollado para la gestión de préstamos de libros, es natural que se presenten fallos en distintos niveles: procesos individuales, comunicación entre sedes, pérdida de mensajes, errores temporales u omisiones. Por esta razón, el sistema fue diseñado teniendo en cuenta mecanismos de detección temprana, recuperación automática y continuidad del servicio, con el fin de garantizar la disponibilidad del sistema y la coherencia eventual de los datos incluso en situaciones adversas.

2.3.1 Tipos de Fallos Considerados

1. Fallos de procesos

Corresponden a la caída inesperada de uno o varios componentes, como el Gestor de Carga (GC), los Actores o el Gestor de Almacenamiento (GA). Estos fallos pueden originarse por errores del propio software, consumo excesivo de recursos o cierres abruptos del proceso. La interrupción afecta directamente el flujo de las operaciones, por lo que el sistema incorpora mecanismos de reintento y failover para minimizar su impacto.

2. Fallos de comunicación

Son inherentes a los entornos distribuidos. Se producen cuando los mensajes intercambiados entre los PS, GC, Actores o entre los GA de cada sede se pierden, se duplican o llegan con retraso. Estos eventos suelen estar relacionados con problemas en la red, congestión de sockets ZeroMQ o interrupciones breves en la conectividad. El sistema contempla timeouts y redireccionamiento para manejar estos escenarios.

3. Fallos de concurrencia

Aparecen cuando múltiples actores intentan modificar el mismo recurso de forma simultánea. Por ejemplo, dos solicitudes de préstamo sobre el mismo libro o una renovación sobre un préstamo ya vencido. Si no se controlan, pueden provocar inconsistencias como préstamos duplicados o estados inválidos. El sistema evita estos problemas mediante validaciones estrictas y control de estados en el GA.

4. Fallos por omisión

Pueden clasificarse en dos tipos:

- **Omisión de envío:** ocurre cuando un PS envía una solicitud pero esta no llega al GC debido a pérdida de mensajes o errores en la conexión.

- **Omisión de recepción:** el mensaje llega al actor correspondiente, pero este no consigue procesarlo o no logra aplicar la actualización en la base de datos.

Ambos casos pueden derivar en operaciones incompletas, por lo que el sistema utiliza reintentos y logs para identificar fallos de este tipo.

5. Fallos de temporización

Ocurren cuando las respuestas superan el tiempo máximo esperado. Esto puede suceder en presencia de saturación de la red, accesos concurrentes a la BD o congestión en los Actores. Para evitar bloqueos prolongados, el sistema incorpora temporizadores (timeouts) que permiten liberar recursos y volver a enviar solicitudes o activar el failover si la sede principal no responde.

2.3.2 Mecanismos

Para garantizar la continuidad operativa del sistema y reducir el impacto de fallos en alguno de sus componentes, se implementaron mecanismos de resiliencia que permiten detectar interrupciones, conmutar hacia componentes funcionales y mantener el estado consistente entre sedes. Estos mecanismos actúan de manera complementaria y aseguran que el servicio continúe disponible incluso ante fallos críticos del Gestor de Almacenamiento o de la base de datos principal.

1. Supervisión continua del Gestor de Almacenamiento (Health-check)

El sistema utiliza un proceso de supervisión que verifica periódicamente el estado del Gestor de Almacenamiento (GA) de la sede principal. Este mecanismo detecta caídas, demoras anormales o ausencia de respuesta mediante señales de heartbeat y timeouts. Cuando el GA primario no responde, se considera que la sede ha fallado y se procede con la recuperación.

2. Conmutación por fallo automática (Failover)

Si el GA principal deja de responder, el sistema activa el mecanismo de **failover**, mediante el cual la sede secundaria —que mantiene una réplica actualizada de los datos— asume el rol principal. Esta transición se realiza sin interrumpir las operaciones en curso, permitiendo que los Procesos Solicitantes sigan enviando solicitudes sin necesidad de conocer qué sede está activa en ese momento.

3. Persistencia temporal mediante archivos de registro (Logs)

Cada operación procesada (préstamos, devoluciones y renovaciones) genera un registro en los archivos del sistema. Estos logs permiten reconstruir la secuencia de eventos en caso de fallos, facilitan tareas de auditoría y soportan la recuperación tras incidentes como reinicios inesperados.

4. Reintento automático de solicitudes fallidas

Cuando el Gestor de Carga detecta que una solicitud no obtuvo respuesta dentro del tiempo establecido, la marca como fallida y activa un mecanismo de reintentos. Esto ayuda a mitigar fallos temporales de comunicación o tiempos de respuesta elevados durante picos de carga.

5. Sincronización periódica y asíncrona entre sedes

La replicación entre GA1 y GA2 se realiza mediante un canal dedicado que envía actualizaciones en segundo plano. Gracias a esta sincronización:

- La sede secundaria mantiene una copia reciente de la base de datos.
- La recuperación ante fallos es más rápida (menor RTO).
- Se reduce significativamente la pérdida de información (bajo RPO).

Como esta sincronización es asíncrona, no afecta el rendimiento de las operaciones en tiempo real.

6. Auditoría y registro de eventos del sistema

Además de los logs de operaciones, el sistema registra fallos detectados, procesos de recuperación, reintentos y anomalías. Esta trazabilidad resulta fundamental para analizar incidentes, realizar pruebas posteriores y evaluar el desempeño del sistema bajo condiciones adversas.

7. Detección de anomalías

El módulo de supervisión también verifica retrasos inusuales, pérdida de mensajes o intentos frecuentes. Este análisis permite identificar síntomas tempranos de degradación del servicio, incluso antes de que ocurra un fallo total.

En el escenario de pruebas de fallos se forzó explícitamente la caída del Gestor de Almacenamiento primario (GA1) en la Sede 1, simulando un fallo crítico del nodo de persistencia principal. El fallo se detectó mediante el mecanismo de health-check y los timeouts configurados en el monitor de salud, que dejaron de recibir respuestas dentro del intervalo esperado. Una vez superado ese umbral, el sistema activó automáticamente el proceso de

failover: GA2 fue promovido a rol principal, comenzó a atender las operaciones entrantes y continuó aplicando las actualizaciones recibidas, garantizando la disponibilidad del servicio sin requerir intervención manual.

2.3.3 Principios de Integridad y Protección

Aunque el proyecto tiene un enfoque académico y no requiere mecanismos avanzados de seguridad, se incorporaron medidas prácticas que garantizan la **integridad de las operaciones**, la **separación de responsabilidades** y la **protección del flujo de mensajes entre procesos**. Estas medidas no introducen sobrecarga adicional al sistema y se ajustan a las capacidades reales del entorno implementado.

1. Separación de responsabilidades entre componentes

Los Procesos Solicitantes no acceden directamente a la base de datos. Todas las operaciones deben pasar por el Gestor de Carga (GC) y los Actores, lo que evita accesos indebidos y protege la integridad de los registros.

2. Validación estricta de operaciones

Antes de procesar cualquier solicitud, el GC y los Actores validan:

- El tipo de operación
- La existencia del libro y usuario
- El estado actual del préstamo
- La coherencia de las fechas y reglas de negocio

Esto evita operaciones inválidas o manipulaciones sobre el estado de un libro.

3. Integridad mediante identificación básica del mensaje

Cada solicitud incluye campos que permiten verificar su validez (tipo de operación, sede de origen, ids). Los mensajes mal formados o incompletos se descartan, evitando inconsistencias.

4. Aislamiento de canales de comunicación

La arquitectura usa canales separados para cada tipo de operación (préstamo, devolución y renovación). Esto reduce el riesgo de interferencias entre procesos y evita que un Actor reciba mensajes que no le corresponden.

5. Trazabilidad mediante registros (logs)

Cada operación queda guardada en los logs del sistema, incluyendo:

- Timestamps
- Tipo de operación
- Resultado
- Sede que la procesó

Esto permite reconstruir la secuencia de eventos ante fallos o auditorías.

3. Diseño del Sistema

3.1 Diagrama de Clases

3.1.1 Diagrama de Clases

El diseño del sistema se basa en un conjunto de clases que representan las entidades principales, los procesos distribuidos y los mensajes que circulan entre las diferentes sedes. Cada clase cumple una responsabilidad claramente definida, de manera que la arquitectura pueda escalar, mantenerse fácilmente y adaptarse a las exigencias de un ambiente distribuido.

Las clases encargadas de manejar solicitudes, procesar operaciones y acceder a la base de datos se encuentran desacopladas entre sí, lo que permite que cada proceso (PS, GC, Actores y GA) se ejecute de manera independiente en diferentes máquinas. La comunicación entre estos procesos distribuidos se realiza mediante ZeroMQ, usando patrones como REQ/REP y PUB/SUB, mientras que la persistencia se maneja a través de una base de datos SQLite replicada entre sedes.

El diagrama de clases refleja la estructura interna del sistema y su relación con los componentes físicos que aparecen en el modelo de despliegue. Las entidades del dominio (Libro, Usuario, Préstamo) se encuentran coordinadas con las clases operativas (SolicitudOperacion, RespuestaOperacion, Actores y Gestor de Almacenamiento), garantizando coherencia entre el modelo de diseño y la implementación real.

3.2 Clases Principales

3.2.1 Clase *ProcesoSolicitante*

La clase *ProcesoSolicitante* representa a un usuario (estudiante, profesor o sistema cliente) que genera solicitudes hacia el sistema distribuido. Cada instancia simula un cliente independiente y puede ejecutar múltiples operaciones de préstamo, devolución o renovación.

Atributos:

- **archivoSolicitudes:** ruta del archivo que contiene la lista de operaciones a ejecutar.

- **socketZeroMQ:** socket REQ utilizado para la comunicación con el Gestor de Carga de su sede.

- **identificadorUsuario:** código del usuario que origina las solicitudes.

- **sedeOrigen:** identifica la sede desde la cual se envía la petición.

- **latencias:** lista donde se registran los tiempos de respuesta para métricas.

Métodos principales:

- **leerArchivo():** lee el archivo de texto que contiene las solicitudes del usuario y las transforma en objetos *SolicitudOperacion*.

- **enviarSolicitud():** envía la solicitud al GC a través del socket ZeroMQ.

- **recibirRespuesta():** recibe una *RespuestaOperacion* generada por GC y la almacena junto con su latencia.

- **registrarLatencia():** almacena los tiempos por operación para cálculo de métricas (p50, p95, p99).

3.2.2 Clase *GestorDeCarga (GC)*

El *Gestor de Carga* es el intermediario entre los Procesos Solicitantes y los Actores. Se encarga de validar las solicitudes, clasificarlas según su tipo y enviarlas al componente correspondiente.

Atributos:

- **socketRequest (REQ/REP):** canal para solicitudes sincrónicas, principalmente préstamos.

- **socketPublisher (PUB):** socket utilizado para publicar eventos de devolución y renovación hacia los Actores.

- **registroLogs:** ruta de archivo donde se registran solicitudes, errores y métricas del sistema.

- **timeoutActor:** tiempo máximo de espera para recibir respuesta en el caso del préstamo.

- **estadoGA:** indica si el Gestor de Almacenamiento principal está disponible o si debe usarse la réplica.

Métodos principales:

- **procesarSolicitud():** interpreta la operación recibida, válida parámetros y decide el método de envío.
- **publicarTopico():** pública la solicitud en el canal correspondiente para devoluciones o renovaciones.
- **reenviarActor():** reenvía solicitudes de préstamo al Actor correspondiente mediante REQ/REP.
- **registrarEvento():** escribe en el archivo de logs un registro de cada operación.
- **conmutarSede():** redirige solicitudes hacia GA2 en caso de failover.

3.2.3 Clase *ActorPrestamo*

El *Actor de Préstamo* procesa las solicitudes sincrónicas que requieren confirmación inmediata. Es el encargado de validar disponibilidad, ejecutar lógica de negocio y actualizar la BD.

Atributos:

- **socketRequest:** socket REP para recibir solicitudes enviadas desde GC.
- **gestorBaseDatos:** referencia al Gestor de Almacenamiento para ejecutar operaciones sobre la base de datos.
- **idActor:** identificador de la instancia (sede 1 o sede2).

Métodos principales:

- **verificarDisponibilidad():** consulta si el libro se encuentra disponible.
- **actualizarStock():** actualiza la base de datos después de aprobar el préstamo.
- **responderGestor():** entrega al GC un mensaje confirmando éxito o fallo.
- **registrarOperacion():** almacena en logs información sobre la solicitud procesada.

3.2.4 Clases *ActorDevolucion* y *ActorRenovacion*

Estos actores procesan operaciones asincrónicas enviadas mediante el patrón PUB/SUB.

Atributos en común:

- **socketSubscriber:** socket SUB que recibe mensajes publicados por el Gestor de Carga.
- **gestorBaseDatos:** referencia al GA para ejecutar cambios en la base de datos.
- **idActor:** identificador del actor.

Métodos principales:

- **devolverLibro():** actualiza el estado del libro a “disponible”.
- **renovarLibro():** extiende el préstamo del usuario según las reglas del sistema.
- **registrarEvento():** escribe información de operación en los logs del Actor.
- **procesarMensaje():** recibe el evento y ejecuta la acción correspondiente.

3.2.5 Clase *GestorDeAlmacenamiento (GA)*

El *Gestor de Almacenamiento* es uno de los componentes más importantes del sistema. Gestiona la base de datos, aplica las operaciones solicitadas y coordina el mecanismo de replicación entre sedes.

Atributos:

- **conexionPrincipal:** conexión activa a la BD de la sede donde opera el GA.
- **socketReplica:** socket PUSH o REQ para enviar cambios a la sede secundaria.
- **socketReplicaIn:** socket PULL o REP para recibir cambios provenientes de la otra sede.
- **estadoServidor:** indicador que especifica si el GA se encuentra activo o en modo secundario.

- **rutaBD:** ubicación de la base de datos SQLite.

Métodos principales:

- **actualizarLibro():** aplica las operaciones de préstamo, devolución y renovación en la base de datos.
- **replicarCambio():** envía actualizaciones hacia la sede secundaria para mantener consistencia eventual.
- **recibirReplica():** recibe y aplica cambios provenientes del GA remoto.
- **activarReplica():** promueve la sede secundaria a principal cuando ocurre un failover.
- **consultarLibro():** obtiene información actualizada para validar solicitudes.

3.2.6 Clase *MonitorDeSalud (HealthCheck)*

El *Monitor de Salud* supervisa continuamente el estado del GA principal y activa la conmutación en caso de detectar fallas graves.

Métodos principales:

- **monitorearServidor():** envía señales periódicas (heartbeats) al GA principal y detecta ausencia de respuesta.
- **activarFailover():** activa la transición hacia el GA réplica para que asuma la operación del sistema.
- **registrarAlerta():** deja constancia en el log de fallos detectados y cambios de estado.

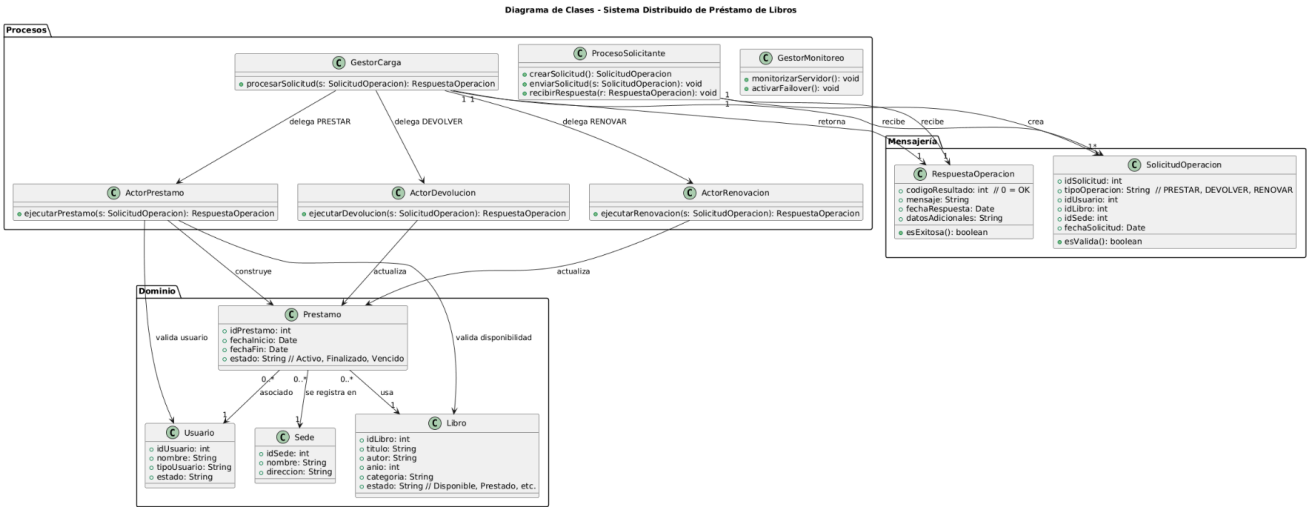


Figura 1. Diagrama de clases del sistema distribuido de préstamo de libros.

3.3 Diagrama de Secuencia

El diagrama de secuencia describe el flujo de mensajes entre los elementos principales del sistema durante la ejecución de una operación típica. En esta entrega se modela el escenario de solicitud de préstamo de un libro.

En la Figura 2 se representa la interacción entre el usuario, el Proceso Solicitante (PS), el Gestor de Carga (GC), el Actor Préstamo (AP) y el Gestor de Almacenamiento (GA). El flujo general es el siguiente:

1. El usuario ingresa los datos del préstamo en la interfaz (`UI Cliente`).

2. El `Proceso Solicitante (PS)` construye una instancia de `SolicitudOperacion` con los datos de la solicitud y la envía al `Gestor de Carga (GC)` mediante el adaptador ZeroMQ cliente (mensaje REQ).

3. El `GC` válida la solicitud y la reenvía al `Actor Préstamo (AP)` como una invocación de operación interna.

4. El `AP` consulta la disponibilidad del libro y del usuario a través del "Gestor de Almacenamiento (GA)`, que a su vez accede a la base de datos local.

5. Si la operación es válida, el `AP` registra el préstamo en la base de datos (por medio del `GA`) y construye una `RespuestaOperacion` indicando éxito, junto con información relevante (por ejemplo, fecha de vencimiento del préstamo).

6. El `GC` envía la `RespuestaOperacion` de vuelta al `PS` usando ZeroMQ (mensaje REP).

7. El `PS` muestra al usuario una confirmación explícita de que la operación fue exitosa o, en caso contrario, el mensaje de error correspondiente.

Es importante resaltar que, aunque el enunciado no lo indique de forma explícita, el diagrama de secuencia incluye siempre un mensaje de confirmación de la operación hacia el cliente, tal como exige la corrección: el usuario debe saber si el préstamo fue aceptado o rechazado.

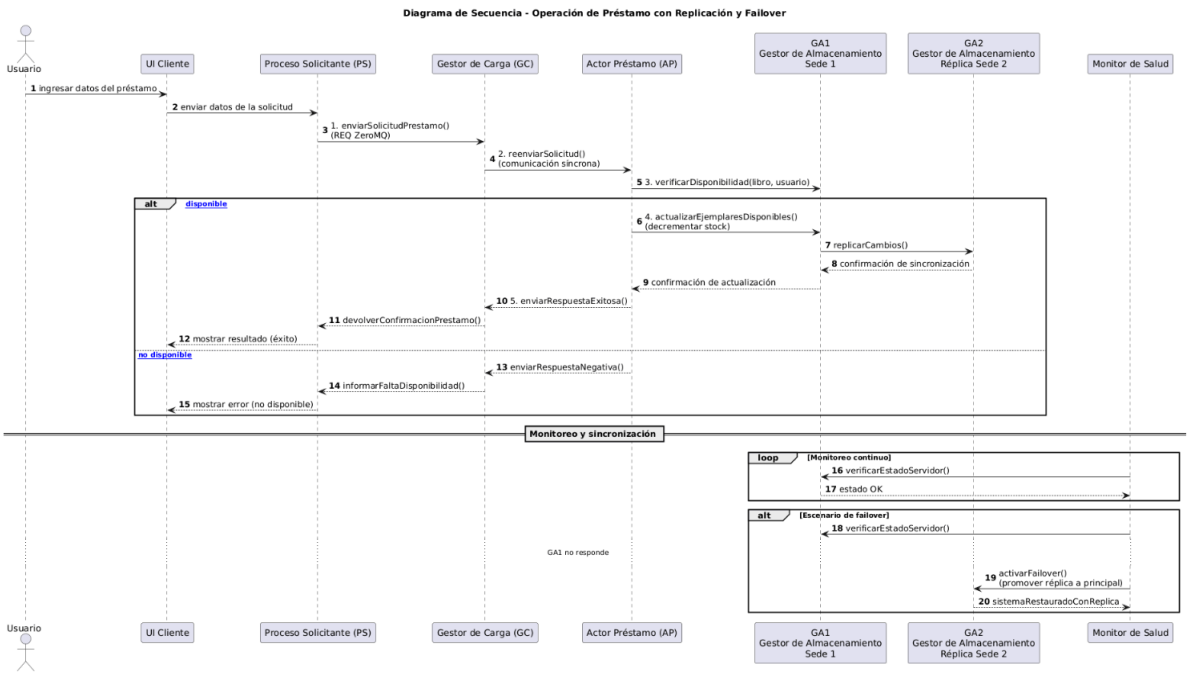


Figura 2. Diagrama de secuencia para la operación de préstamo de un libro.

3.4 Diagrama de Componentes

El diagrama de componentes detalla los módulos lógicos que conforman el sistema distribuido de préstamo de libros y las relaciones entre ellos. Este diagrama es una extensión del diagrama de despliegue: mientras el despliegue muestra en qué máquinas se ejecuta el sistema, el diagrama de componentes muestra qué piezas de software existen en cada nodo y cómo se comunican a nivel de interfaces.

A nivel lógico, el sistema se organiza en los siguientes grupos de componentes:

Cliente

- Componente `UI Cliente`: gestiona la interacción con el usuario final.
- Componente `Proceso Solicitante (PS)`: construye las solicitudes de operación (préstamo, devolución, renovación) y procesa las respuestas.
- Componente `Adaptador ZeroMQ Cliente`: encapsula los detalles de comunicación con el servidor mediante ZeroMQ (REQ/REP).

Servidor de sede

- Componente `Gestor de Carga (GC)`: expone la interfaz de servicio a los PS, valida las solicitudes y las enruta al actor de negocio correspondiente.
- Componentes `Actor Préstamo`, `Actor Devolución`, `Actor Renovación`: encapsulan la lógica de negocio de cada tipo de operación.

- Componente `Gestor de Almacenamiento (GA)`: ofrece operaciones de alto nivel sobre la información de libros, usuarios y préstamos, y orquesta el acceso a la base de datos y la replicación.

Persistencia

- Componente `Adaptador SQLite`: implementa las operaciones CRUD sobre la base de datos local.
- Componente `BD SQLite`: representa el motor de base de datos en cada sede.

Replicación entre sedes

- Componente `Canal de Replicación GA1–GA2`: encapsula el mecanismo de envío y recepción de actualizaciones entre los gestores de almacenamiento de cada sede para mantener la consistencia eventual.

En cada sede se despliega una instancia de este conjunto de componentes (GC, actores, GA, adaptador y BD), tal como se muestra en el diagrama de despliegue de la Figura 3. El diagrama de componentes se centra en las dependencias de software y las interfaces expuestas entre ellos.

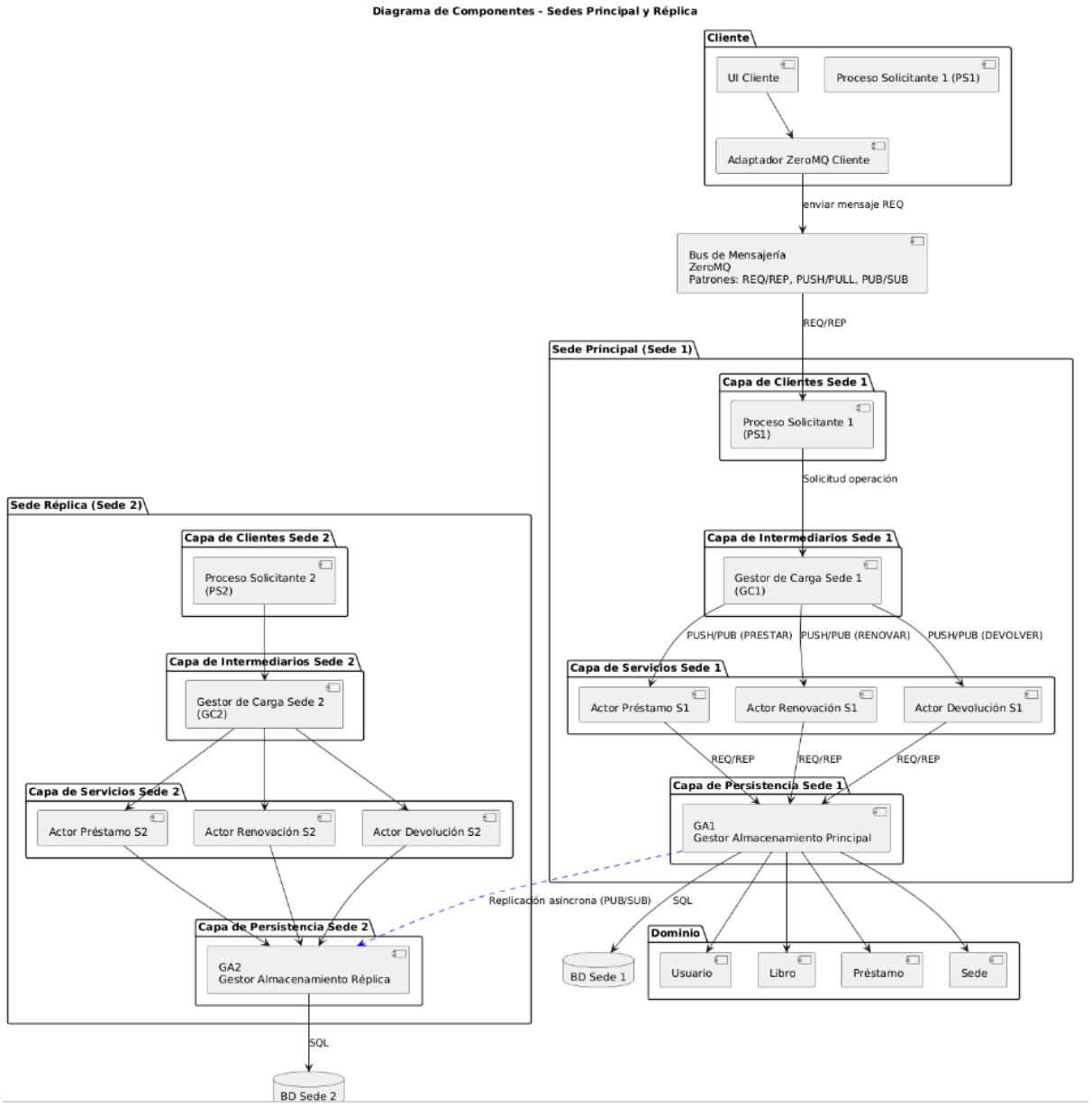


Figura 3. Diagrama de componentes del sistema distribuido de préstamo de libros.

En cuanto a la trazabilidad de diseño, cada clase crítica del diagrama de clases (Sección 3.1) tiene su correspondencia directa como componente en el diagrama de componentes. Por ejemplo, las clases ProcesoSolicitante, GestorDeCarga, ActorPrestamo, ActorDevolucion, ActorRenovacion y GestorDeAlmacenamiento se proyectan respectivamente en los componentes homónimos. De esta forma, la vista de componentes extiende la definición del diagrama de despliegue, evitando inconsistencias entre el modelo lógico (clases), la estructura de software (componentes) y la distribución física en máquinas.

3.5 Diagrama de Despliegue

El sistema se desplegó en tres máquinas distribuidas, cada una cumpliendo un rol específico dentro de la arquitectura:

Máquina 1 – PC Local (Juan Camargo)

Componentes:

- Gestor de Carga – Sede 1 (GC1)
- Actores de Sede 1 (Préstamo, Devolución, Renovación)
- Procesos Solicitantes (PS1)

Máquina 2 – VM Universidad (IP 10.43.102.221)

Componentes:

- Gestor de Almacenamiento – Sede 1 (GA1)
- Base de Datos de Sede 1 (SQLite)

Máquina 3 – VM Universidad (IP 10.43.102.23)

Componentes:

- Gestor de Carga – Sede 2 (GC2)
- Actores de Sede 2
- Gestor de Almacenamiento – Sede 2 (GA2)
- Base de Datos de Sede 2 (SQLite)

Nodo	Componente	Descripción
Sede 1 (Equipo 1)	GC1, Actores (Préstamo, Devolución, Renovación), GA1, BD Primaria	Biblioteca sede norte. Equipo donde se procesa la base de datos principal y se atienden operaciones prioritarias.
Sede 2 (Equipo 2)	GC2, Actores, GA2, BD Réplica	Biblioteca sede sur. Mantiene una réplica actualizada y asume el rol principal en caso de fallo del GA de la sede 1.

Sede 3 (Equipo 3)	Procesos Solicitantes (PS)	Equipo utilizado para simular usuarios reales del sistema, ejecutar archivos de operaciones y medir tiempos de respuesta.

El diagrama de despliegue muestra cómo se distribuyen físicamente los componentes del sistema en las máquinas que conforman la red. El sistema se ejecuta sobre tres computadoras, que representan las sedes físicas de la biblioteca y los clientes solicitantes:

- **Sede 1:** aloja el Gestor de Carga (GC1), los Actores encargados de las operaciones de préstamo, devolución y renovación, el Gestor de Almacenamiento (GA1) y la base de datos local.
- **Sede2:** Replica la estructura de la sede 1, con su propio Gestor de Carga (GC2), sus Actores asociados, un Gestor de Almacenamiento (GA2) y una base de datos local que mantiene una copia secundaria de la información.
- **Cliente (Sede 3):** Ejecuta los Procesos Solicitantes (PS), que simulan a los usuarios finales y generan las solicitudes de servicio hacia los Gestores de Carga de cada sede.

Cada sede cuenta con su propia base de datos SQLite local, que mantiene una copia sincronizada mediante un proceso de replicación administrado por los Gestores de Almacenamiento (GA). Las comunicaciones entre los procesos se realizan mediante ZeroMQ sobre TCP/IP, implementando:

- Una topología *peer-to-peer* entre los Gestores de Carga y los Actores, y
- Un esquema cliente–servidor entre los Procesos Solicitantes y los Gestores de Carga.

Este despliegue asegura que el sistema mantenga disponibilidad y autonomía operativa en cada sede, incluso ante fallos parciales o desconexiones de red.

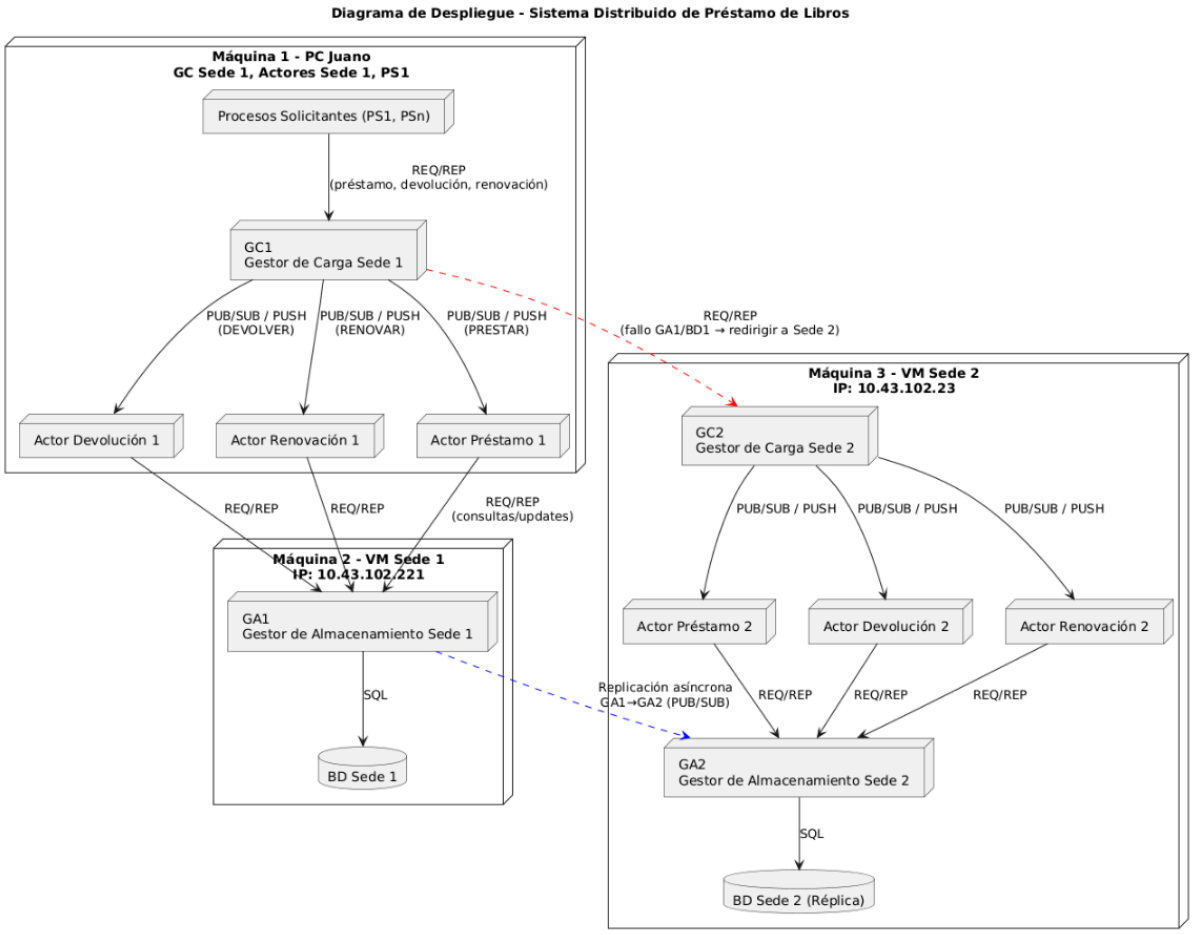


Figura 4: Diagrama de despliegue del sistema de préstamo de libros

4. Protocolo de Pruebas

El protocolo de pruebas tiene como propósito validar el funcionamiento, desempeño, tolerancia a fallos y consistencia del Sistema Distribuido de Préstamo de Libros bajo diferentes condiciones operativas. Estas pruebas buscan comprobar la capacidad del sistema para manejar solicitudes concurrentes, mantener sincronizada la información entre sedes y recuperarse automáticamente ante la caída del servidor principal.

4.1 Objetivos de las Pruebas

El propósito de las pruebas es evaluar de manera integral el comportamiento del Sistema Distribuido de Préstamo de Libros bajo diferentes condiciones operativas, verificando que cada uno de sus componentes funcione de forma correcta, estable y coherente. En primera instancia, las pruebas buscan confirmar que el sistema es capaz de procesar solicitudes simultáneas de préstamo, devolución y renovación, manteniendo la consistencia de los datos mediante el proceso de replicación entre GA1 y GA2.

Asimismo, las pruebas permiten validar la disponibilidad del sistema ante fallos, particularmente frente a la caída del Gestor de Almacenamiento principal. Para ello, se comprueba la correcta activación del mecanismo de failover y la continuidad del servicio

durante la conmutación hacia la sede secundaria, asegurando que los usuarios no perciban interrupciones en la operación.

Otro objetivo central consiste en analizar el rendimiento del sistema bajo diferentes niveles de carga, midiendo métricas clave como latencia, throughput, distribución de tiempos (p50, p95 y p99), desviación estándar y tasa de errores. Esto facilita determinar la capacidad del sistema para operar eficientemente en escenarios de alta concurrencia y bajo estrés progresivo (4, 6 y 10 usuarios simultáneos).

Asimismo, las pruebas buscan evaluar la estabilidad de los patrones de comunicación ZeroMQ utilizados: REQ/REP para operaciones sincrónicas y PUB/SUB para operaciones asincrónicas. Con ello se pretende identificar comportamientos ante congestión del canal, tiempos de espera prolongados, fallos de red, omisiones o pérdida de mensajes, evaluando la robustez del sistema en escenarios adversos.

Finalmente, se valida que todos los procesos distribuidos —Procesos Solicitantes, Gestores de Carga, Actores y Gestores de Almacenamiento— interactúen correctamente, registren los eventos relevantes, preserven la integridad de los datos y mantengan un flujo de ejecución coherente. De esta forma, las pruebas permiten confirmar que la arquitectura implementada cumple con los principios fundamentales de los sistemas distribuidos: disponibilidad, tolerancia a fallos, consistencia eventual y estabilidad operativa.

4.2 Pruebas de Estrés y Carga

Para complementar las pruebas manuales realizadas con Procesos Solicitantes tradicionales, se empleó **Locust**, una herramienta especializada en la simulación de múltiples usuarios concurrentes y en la medición del rendimiento de sistemas distribuidos. Su uso permitió evaluar la arquitectura bajo condiciones controladas de carga progresiva, reflejando el comportamiento real del sistema en situaciones de alta demanda.

1. Adaptación del sistema para Locust

Dado que el sistema no expone endpoints HTTP de forma nativa, fue necesario implementar un **gateway HTTP–ZMQ mediante Flask**, el cual actúa como intermediario entre Locust y la arquitectura distribuida real. Este gateway se encarga de:

- Recibir las solicitudes HTTP generadas por Locust.
- Transformarlas en objetos *SolicitudOperacion*.
- Enviarlas al Gestor de Carga mediante el patrón **REQ/REP** de ZeroMQ.

- Recibir la *RespuestaOperacion*.
- Retornar la respuesta como JSON para su registro en Locust.

Esta adaptación permite ejecutar pruebas de carga sin modificar la lógica interna del sistema y sin alterar los flujos reales de comunicación entre PS, GC, Actores y GA.

2. Métricas registradas por Locust

Durante las pruebas, Locust recopiló las métricas fundamentales para evaluar el desempeño y estabilidad del sistema:

- **Latencia por operación (ms)**
- **Requests por segundo (RPS)**
- **Distribución de tiempos de respuesta**
- **Tasa de éxito y error**
- **Comportamiento del sistema bajo saturación**

Estas métricas permiten identificar cómo evoluciona el rendimiento y la estabilidad del sistema a medida que aumentan las solicitudes concurrentes.

3. Escenarios evaluados

Locust se ejecutó en tres escenarios de concurrencia, cada uno representando diferentes niveles de estrés sobre la arquitectura distribuida:

- **4 usuarios simultáneos**
- **6 usuarios simultáneos**
- **10 usuarios simultáneos**

En cada escenario se evaluaron las operaciones principales del sistema:

- **PRESTAR**, operación sincrónica gestionada mediante **REQ/REP**.
- **RENOVAR**, operación asincrónica manejada mediante **PUB/SUB**.
- **DEVOLVER**, también gestionada mediante **PUB/SUB**.

Asimismo, se registraron métricas globales de rendimiento, entre ellas:

- **Throughput general** (solicitudes procesadas por segundo)
- **Tiempo de respuesta promedio (ms)**
- **Desviación estándar**, como indicador de estabilidad
- **Distribución completa de tiempos**, necesaria para comprender el comportamiento del sistema bajo carga real

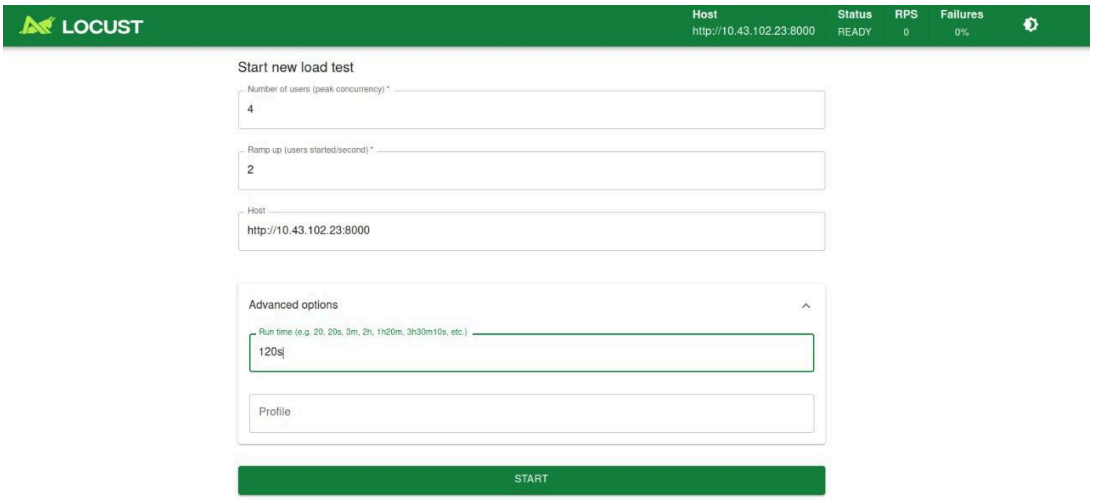
Estos escenarios permiten comparar la capacidad del sistema para escalar horizontalmente, identificar cuellos de botella y validar la estabilidad del flujo de comunicación entre los componentes distribuidos.

4. Duración y parámetros de ejecución

Todas las pruebas se ejecutaron durante 120 segundos (2 minutos), con un spawn rate de 2 usuarios por segundo, lo que generó una rampa de carga gradual y controlada. Esto permitió observar con precisión cómo se comporta el sistema a medida que aumenta la concurrencia y cómo evolucionan las métricas durante el periodo de carga constante.

En las siguientes imágenes se evidencia las pruebas que realizamos

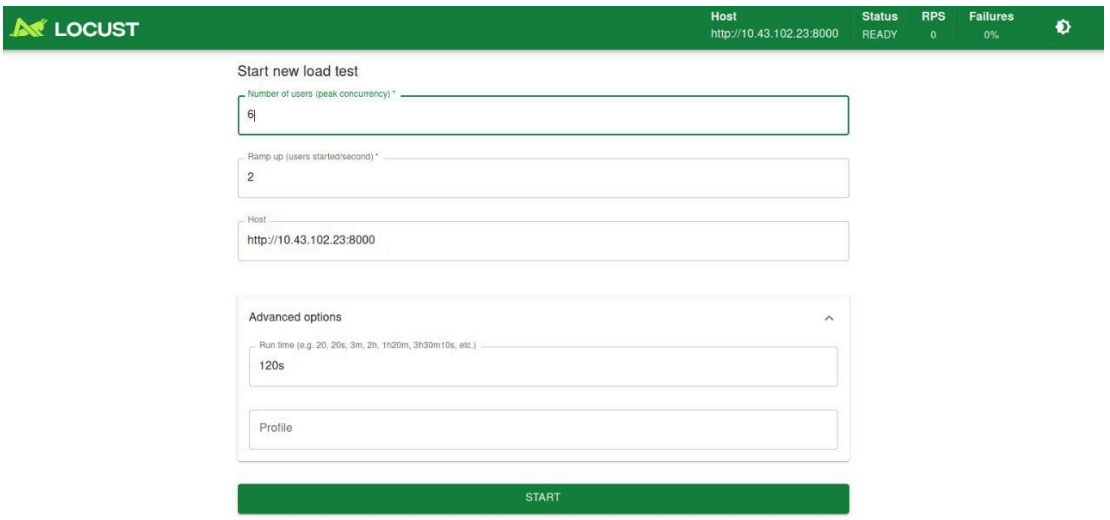
Para el experimento inicial con 4 usuarios



(Figura 5. Configuración de prueba en Locust con 4 usuarios).

Se configuró una carga inicial de 4 usuarios con un ramp-up de 2 usuarios por segundo y un tiempo total de ejecución de 120 segundos.

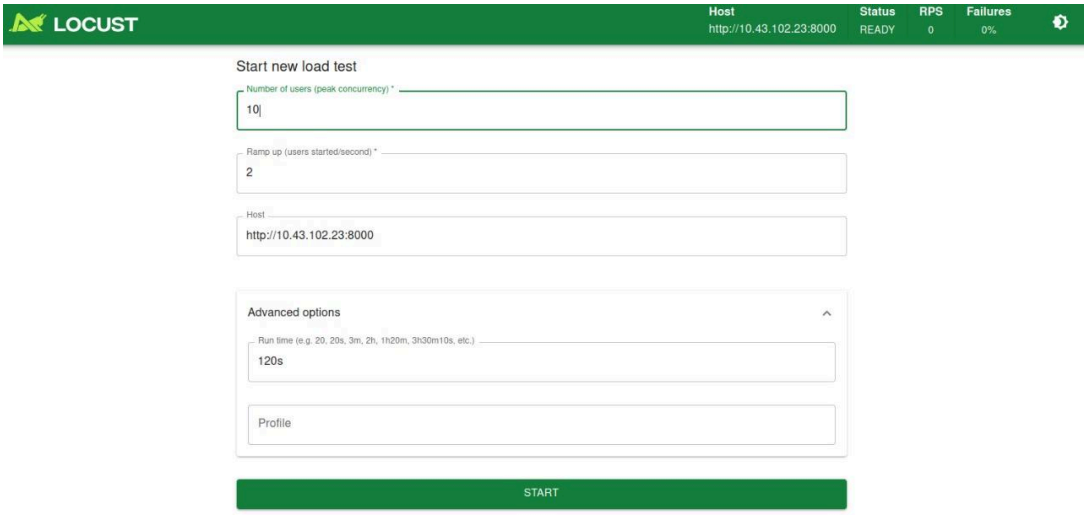
Para el segundo experimento con 6 usuarios



(Figura 6. Configuración de prueba en Locust con 6 usuarios)

Se configuró una carga inicial de 6 usuarios con un ramp-up de 2 usuarios por segundo y un tiempo total de ejecución de 120 segundos.

Para el tercer experimento con 10 usuarios

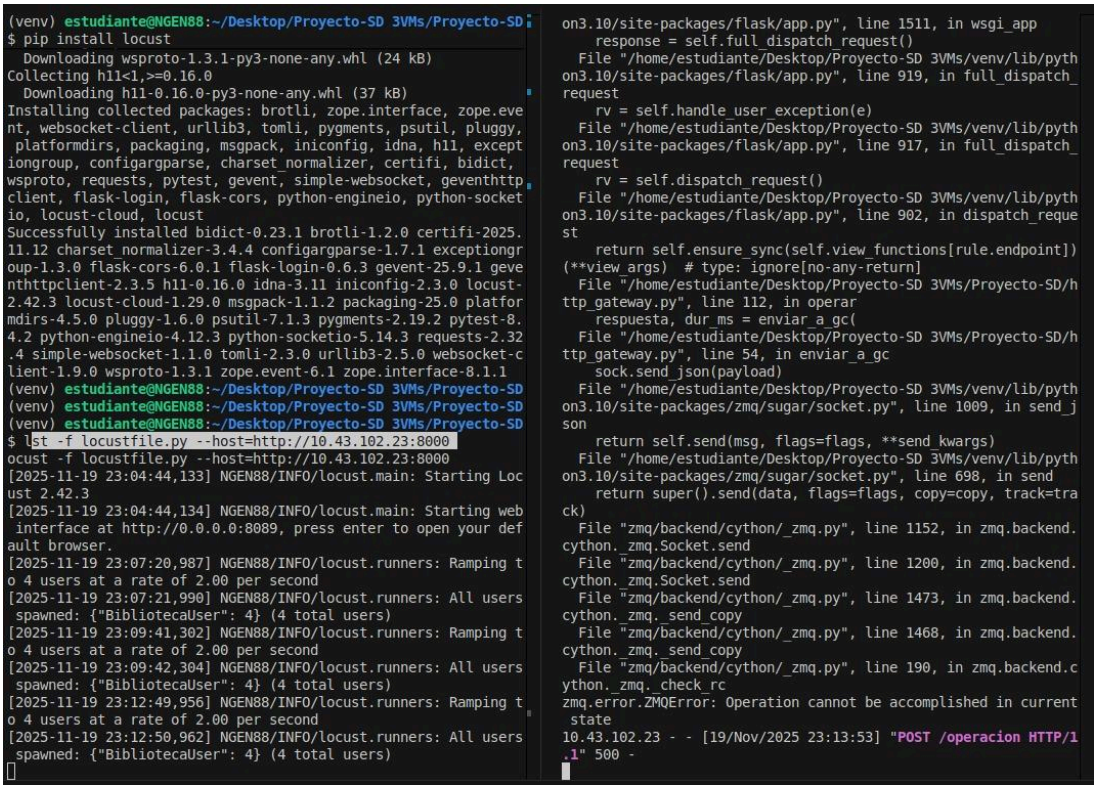


(Figura 7. Configuración de prueba en Locust con 10 usuarios)

Se configuró una carga inicial de 10 usuarios con un ramp-up de 2 usuarios por segundo y un tiempo total de ejecución de 120 segundos.

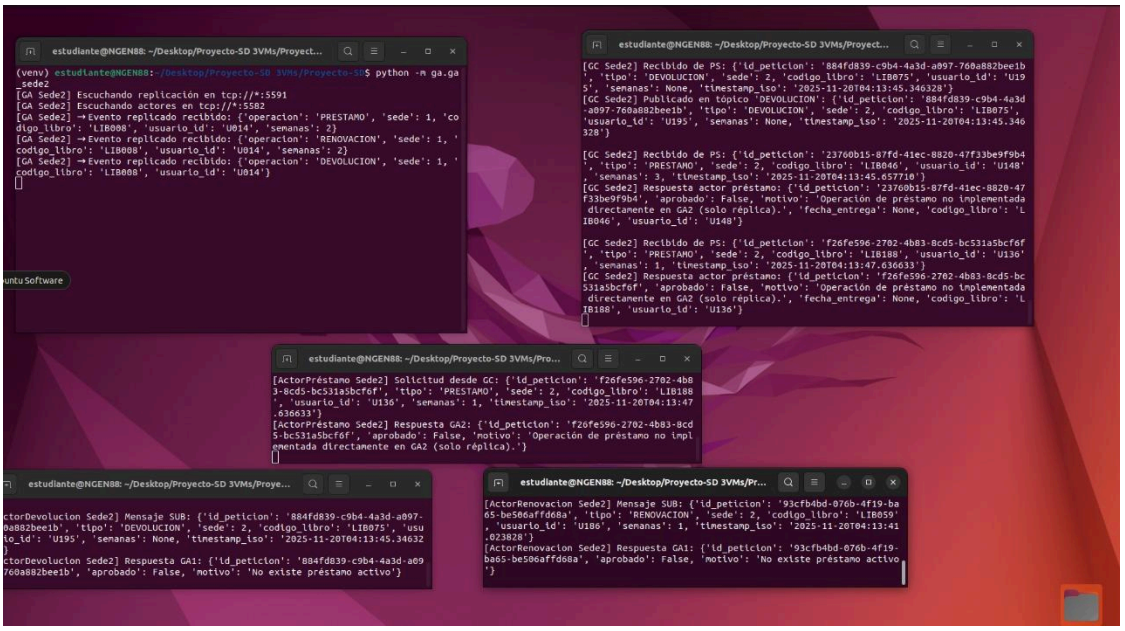
A continuación, se presenta la evidencia del funcionamiento de las pruebas de carga realizadas con Locust, utilizando el gateway HTTP–ZMQ implementado específicamente para este sistema.

- **4 Usuarios**



(Figura 8. Ejecución en consola de la prueba con 4 usuarios)

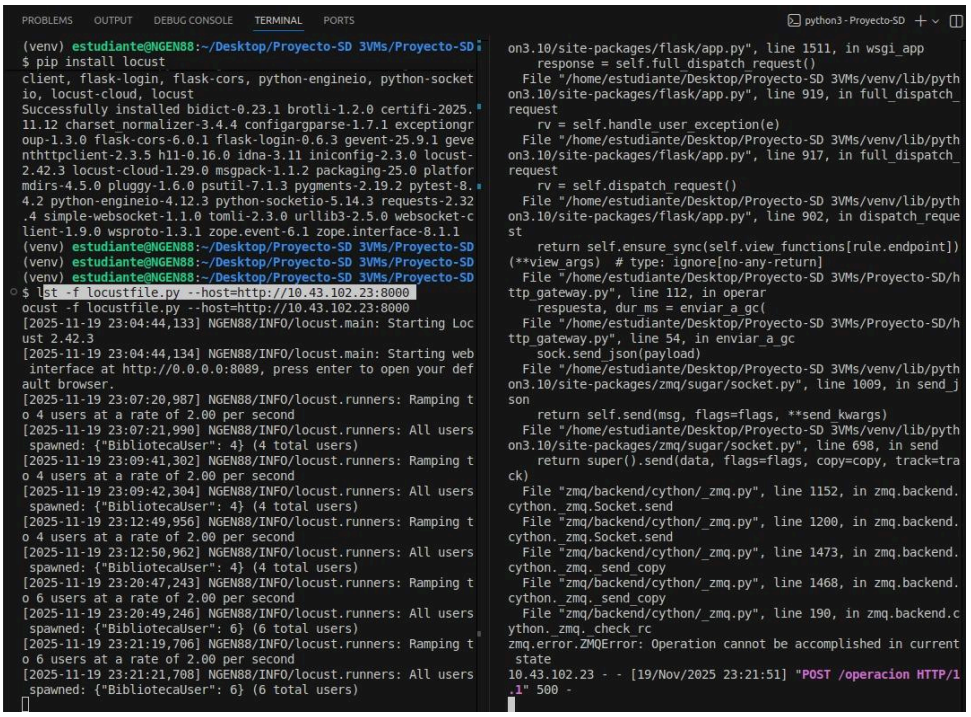
Se observa la ejecución de Locust generando solicitudes hacia el gateway HTTP–ZMQ y los mensajes de log del backend durante la prueba de 4 usuarios.



(Figura 9. Nodos de backend atendiendo la prueba con 4 usuarios)

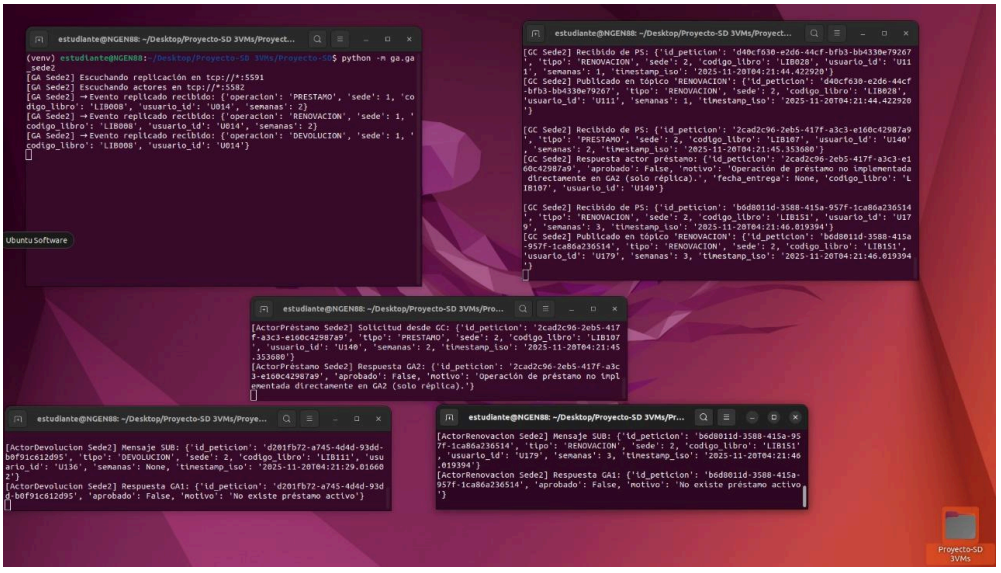
Se visualizan las distintas instancias del backend procesando las operaciones de biblioteca disparadas por los 4 usuarios concurrentes.

- **6 Usuarios**



(Figura 10. Ejecución en consola de la prueba con 6 usuarios)

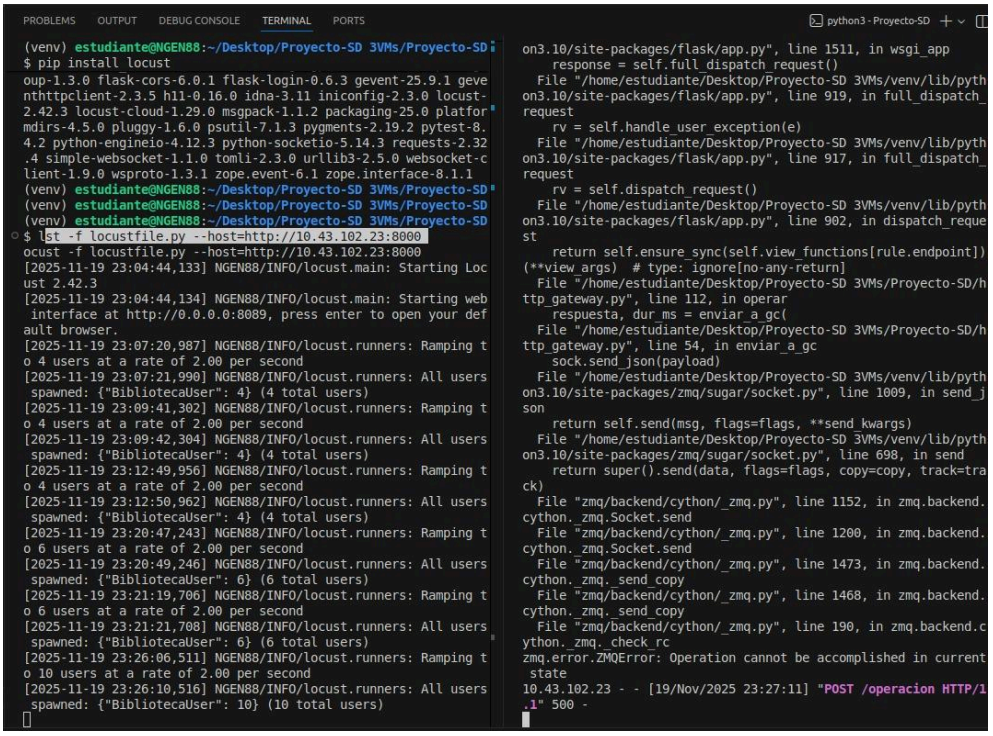
Se muestran los logs de Locust y del backend para la prueba de 6 usuarios, donde se incrementa la frecuencia de solicitudes y los eventos de error.



(Figura 11. Nodos de backend atendiendo la prueba con 6 usuarios)

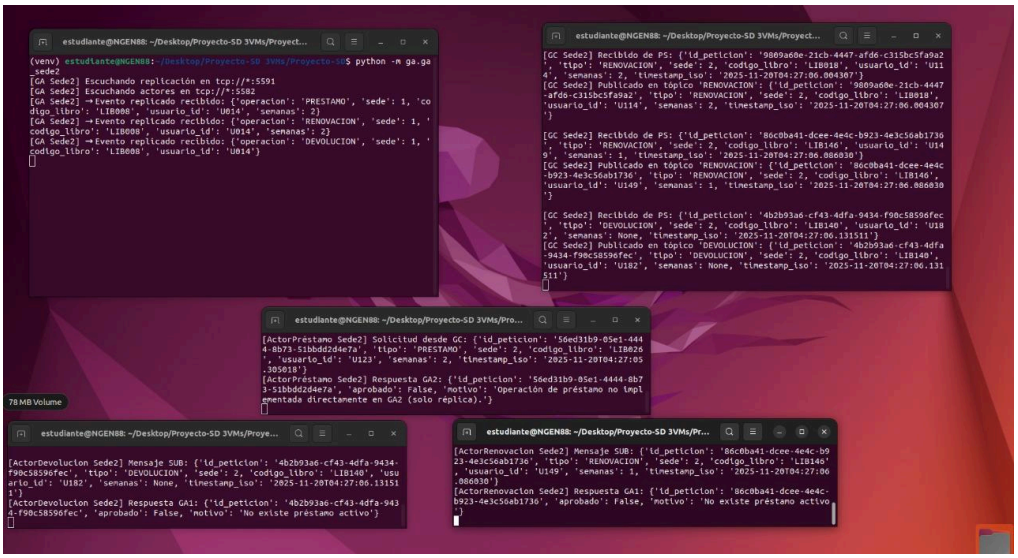
Se evidencia cómo las diferentes instancias del backend procesan de forma paralela las operaciones generadas por los 6 usuarios concurrentes.

- **10 Usuarios**



(Figura 12. Ejecución en consola de la prueba con 10 usuarios)

Se visualiza el aumento de solicitudes concurrentes, junto con mensajes de error más frecuentes debido a la saturación del gateway HTTP–ZMQ.



(Figura 13. Nodos de backend atendiendo la prueba con 10 usuarios)

Se aprecia la actividad simultánea de las diferentes instancias del backend para soportar la carga máxima de 10 usuarios concurrentes.

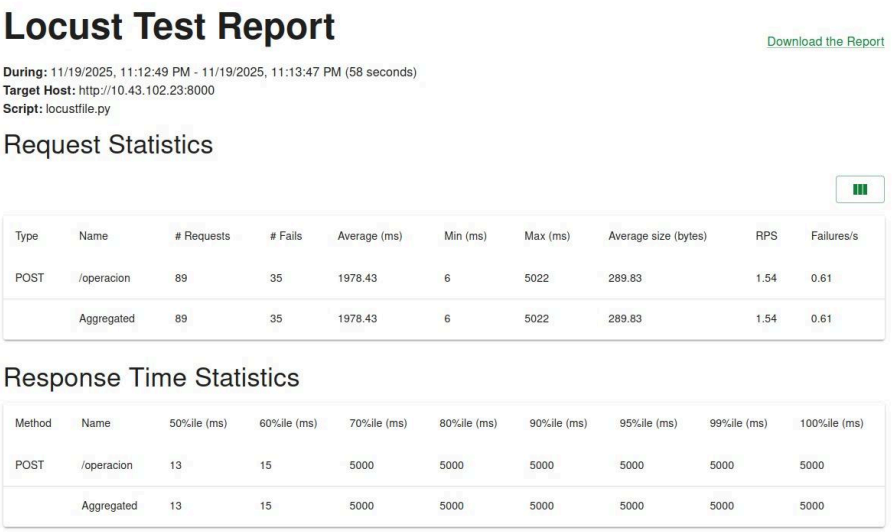
Cuadro 1. Escenario de pruebas utilizado (Experimento II)

Elemento	Descripción
Escenario	Experimento II – Pruebas de carga con Locust
Máquinas	3 máquinas: GC/Actores, GA1–GA2 y cliente (PS/Locust)
Replicación	Replicación asíncrona GA1 → GA2 entre sedes
Usuarios concurrentes	4, 6 y 10 usuarios simulados en Locust
Duración	120 segundos por escenario, con ramp-up de 2 usuarios/s

4.3 Prueba de Carga

Prueba de carga realizada con Locust.

- **4 Usuarios**



Locust Test Report

Download the Report

During: 11/19/2025, 11:12:49 PM - 11/19/2025, 11:13:47 PM (58 seconds)
Target Host: http://10.43.102.23:8000
Script: locustfile.py

Request Statistics

Type	Name	# Requests	# Fails	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	RPS	Failures/s
POST	/operacion	89	35	1978.43	6	5022	289.83	1.54	0.61
	Aggregated	89	35	1978.43	6	5022	289.83	1.54	0.61

Response Time Statistics

Method	Name	50%ile (ms)	60%ile (ms)	70%ile (ms)	80%ile (ms)	90%ile (ms)	95%ile (ms)	99%ile (ms)	100%ile (ms)
POST	/operacion	13	15	5000	5000	5000	5000	5000	5000
	Aggregated	13	15	5000	5000	5000	5000	5000	5000

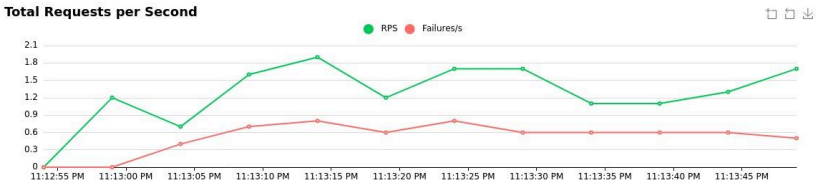
(*Figura 14. Reporte inicial de Locust 4 usuarios)*

Se observan las estadísticas generales de requests, fallas, tiempos promedio y percentiles para la prueba de 4 usuarios.

Failures Statistics

# Failures	Method	Name	Message
35	POST	/operacion	HTTPError('500 Server Error: INTERNAL SERVER ERROR for url: /operacion')

Charts



(Figura 15. Gráfica de RPS y fallos por segundo – 4 usuarios)

El gráfico evidencia un RPS estable alrededor de 1.5 -- 1.8 y un nivel moderado de fallas debido a timeouts en el gateway HTTP–ZMQ.

(Figura 17. Percentiles p50 y p95 – 4 usuarios)

El p50 se mantiene bajo (~13 ms), mientras que el p95 alcanza los 5000 ms por congestión en el patrón REQ/REP.

En base a las imágenes esta prueba de 4 usuarios representa una carga ligera, donde se evalúa el funcionamiento natural del sistema sin congestión elevada.

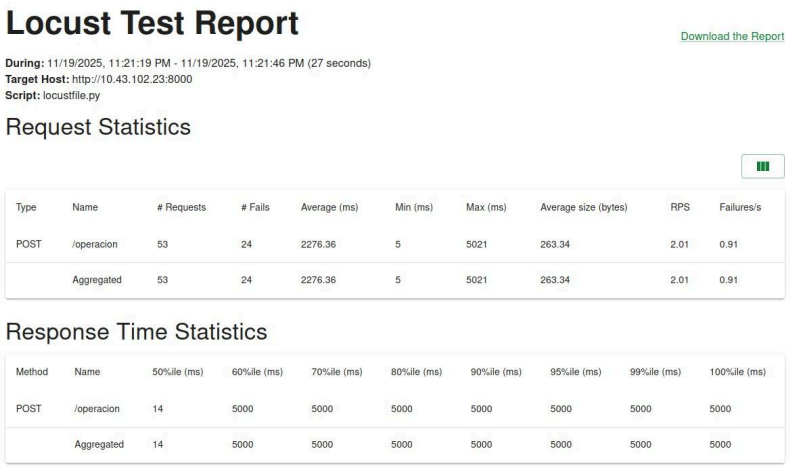
Resultados principales

- Requests totales: 89
- Fallos: 35
- RPS promedio: 1.54
- Latencia promedio: 1978 ms
- p50: 13 ms
- p95–p99: 5000 ms

Interpretación: El sistema opera de manera estable y rápida cuando no hay espera en cola:

- El p50 demuestra tiempos excelentes (<15 ms).
- El p95, sin embargo, indica que bajo ocupación del GC, las operaciones pueden tardar hasta 5 segundos, evidenciando el comportamiento bloqueante del socket REQ/REP.

- **6 Usuarios**



Locust Test Report

Download the Report

During: 11/19/2025, 11:21:19 PM - 11/19/2025, 11:21:46 PM (27 seconds)
Target Host: http://10.43.102.23:8000
Script: locustfile.py

Request Statistics

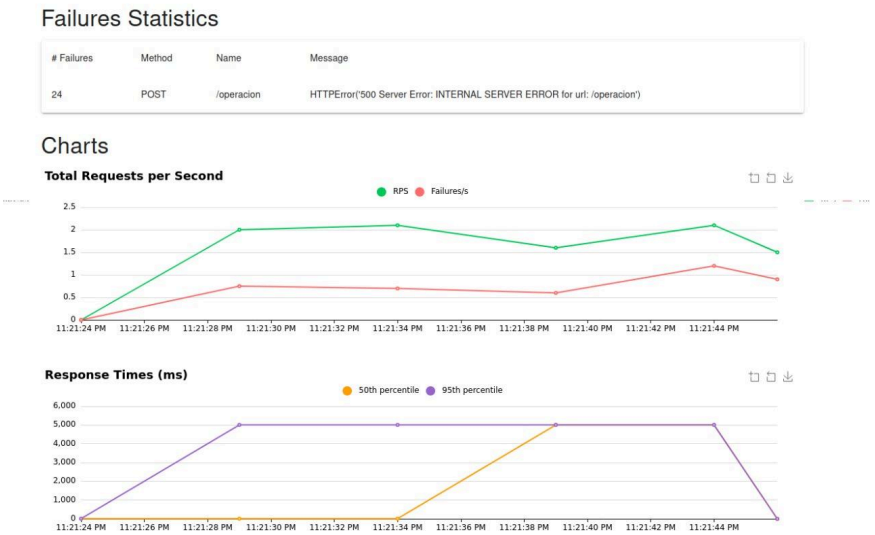
Type	Name	# Requests	# Fails	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	RPS	Failures/s
POST	/operacion	53	24	2276.36	5	5021	263.34	2.01	0.91
	Aggregated	53	24	2276.36	5	5021	263.34	2.01	0.91

Response Time Statistics

Method	Name	50%ile (ms)	60%ile (ms)	70%ile (ms)	80%ile (ms)	90%ile (ms)	95%ile (ms)	99%ile (ms)	100%ile (ms)
POST	/operacion	14	5000	5000	5000	5000	5000	5000	5000
	Aggregated	14	5000	5000	5000	5000	5000	5000	5000

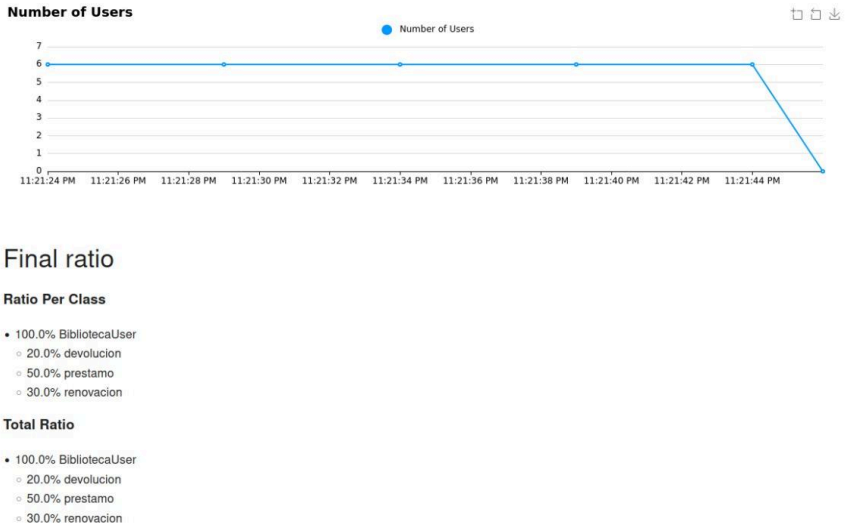
(Figura 18. Reporte de Locust 6 usuarios)

Se presentan las estadísticas consolidadas de requests, tasas de error, tiempos de respuesta promedio y percentiles para la prueba con 6 usuarios.



(Figura 19. Gráfica de RPS y fallos por segundo 6 usuarios)

El gráfico refleja un incremento del RPS respecto a la prueba de 4 usuarios y un aumento de las fallas asociadas a timeouts y errores internos del servidor.



(Figura 20. Percentiles p50 y p95 – 6 usuarios)

Se observa que el p50 se mantiene en valores bajos, mientras que el p95 se mantiene cercano al límite superior configurado (5000 ms), evidenciando cola de peticiones retrasadas.

En base a las imágenes de 6 usuarios, se agrega presión adicional sobre el GC y los actores.

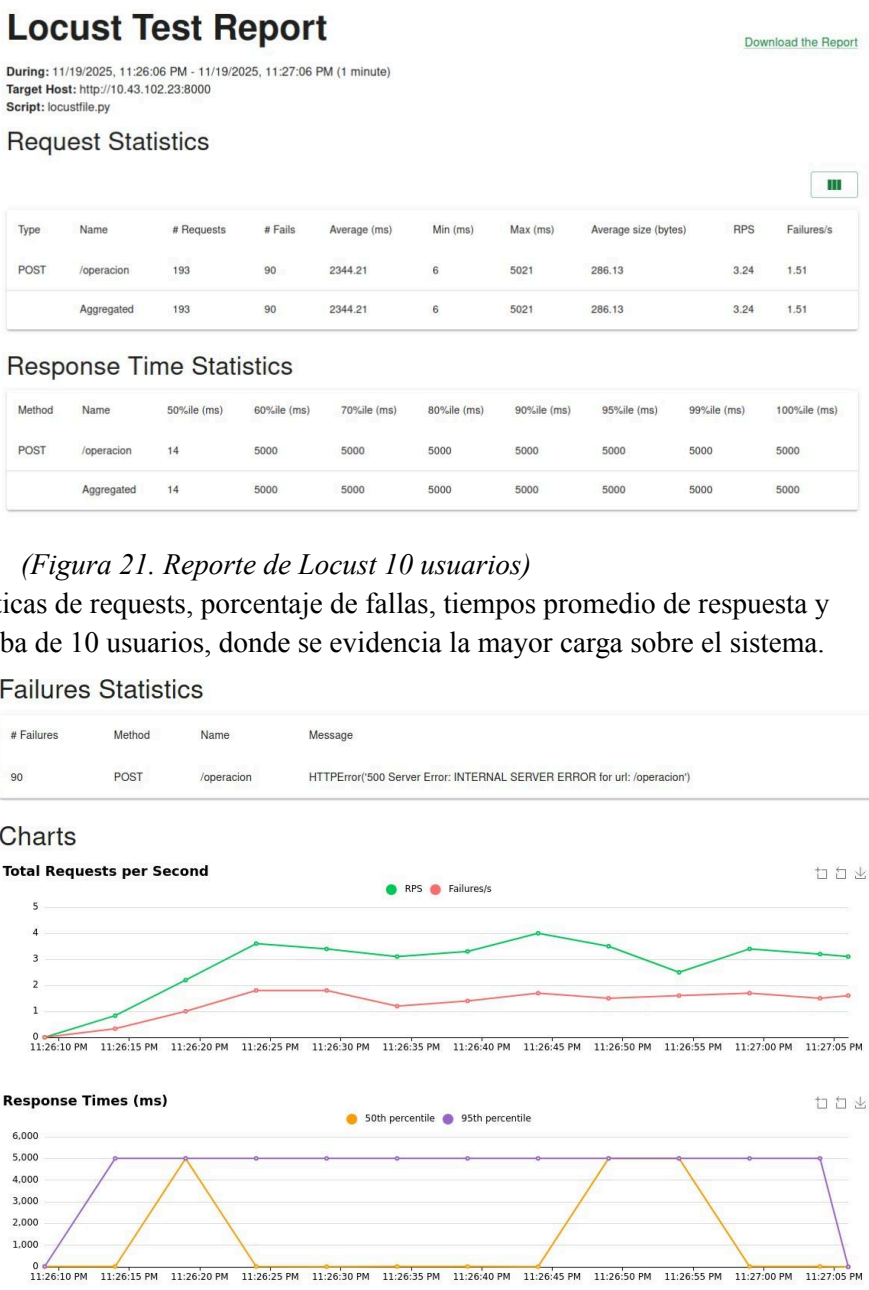
Resultados principales

- Requests totales: 53
- Fallos: 24
- RPS: 2.01
- Latencia promedio: 2276 ms
- p50: 14 ms
- p95: 5000 ms

Interpretación:

- El throughput sube (de 1.5 a 2.0 RPS).
- No hay caída del sistema; mantiene operación estable.
- El p50 continúa siendo excelente.
- El p95 sigue saturado al límite de timeout, lo cual es consistente con la arquitectura asíncrona REQ/REP.

- **10 usuarios**

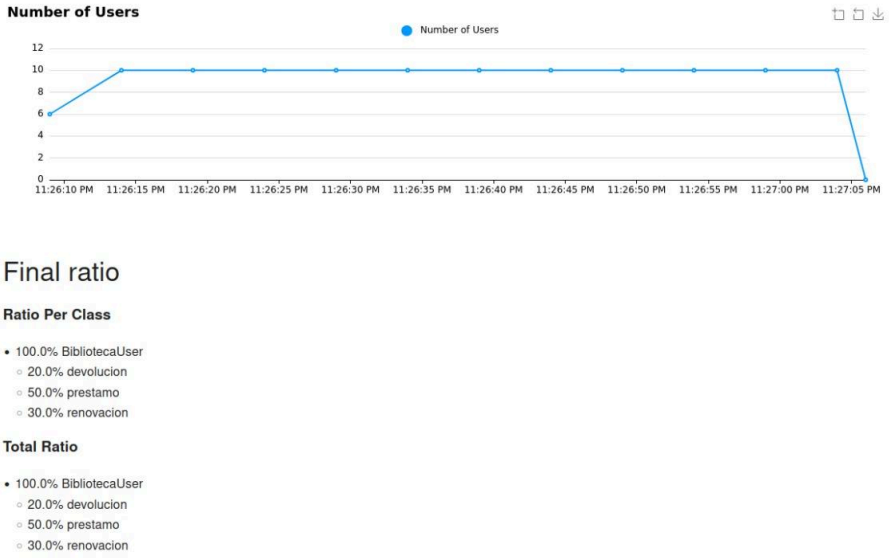


(Figura 21. Reporte de Locust 10 usuarios)

Se muestran las estadísticas de requests, porcentaje de fallas, tiempos promedio de respuesta y percentiles para la prueba de 10 usuarios, donde se evidencia la mayor carga sobre el sistema.

(Figura 22. Gráfica de RPS y fallos por segundo – 10 usuarios)

El gráfico indica un RPS superior al de las pruebas anteriores, acompañado de un incremento significativo en el número de fallas por timeouts y errores internos.



(Figura 23. Percentiles p50 y p95 – 10 usuarios)

Aunque el p50 permanece bajo, el p95 se mantiene en el valor máximo configurado (5000 ms), lo que indica que parte importante de las solicitudes experimenta alta latencia bajo esta carga.

En base a las imágenes de 10 usuarios, se ve la máxima carga exigida por el enunciado.

Resultados principales

- Requests totales: 193
- Fallos: 90
- RPS: 3.24
- Latencia promedio: 2344 ms
- p50: 14 ms
- p95/p99: 5000 ms

Interpretación:

- El sistema escala en throughput (1.54 → 2.01 → 3.24).
- Procesa casi 200 solicitudes en 60 segundos.
- Los fallos aumentan porque el gateway entra en timeout cuando REQ/REP se congestiona.
- El sistema no colapsa, sigue respondiendo y sincronizando GA2 desde GA1.

Explicación de Percentiles:

- El p95 = 5000ms muestra un **timeout del lado del GC**.
- El p50 bajo (13–14 ms) muestra que bajo carga baja el sistema es rápido.
- La brecha entre p50 y p95 demuestra saturación del pipeline.

4.3.1 Comparación Global entre Escenarios

- **Evolución del Throughput (RPS)**

Usuarios	RPS
4	1.54
6	2.01
10	3.24

El sistema escala linealmente, lo cual indica buena capacidad de procesamiento.

- **Latencia (p50 vs p95)**

Usuarios	p50	p95
4	13 ms	5000 ms
6	14 ms	5000 ms
10	14 ms	5000 ms

  - El **p50 demuestra excelente rendimiento**,
  - El **p95 permanece alto** porque REQ/REP bloquea hasta recibir respuesta.

- **Fallos**

Usuarios	Fallos	Motivo
4	35	Timeout REQ/REP
6	24	Timeout REQ/REP
10	90	Congestión REQ/REP

  - Los fallos NO se deben a caída de sedes ni problemas de replicación.
  - Son consecuencia del gateway HTTP–ZMQ cuando REQ/REP queda bloqueado.

4.3.2 Tabla Rendimiento

Métrica	4 usuarios	6 usuarios	10 usuarios
Request Totales	89	53	193

RPS	1.54	2.01	3.24
Fallos	35	24	90
Latencia p50	13 ms	14 ms	14 ms
Latencia p95	5000 ms	5000 ms	5000 ms
Error Principal	500 Timeout	500 Timeout	500 Timeout

La Tabla 4 resume el comportamiento del sistema bajo los tres niveles de carga evaluados (4, 6 y 10 usuarios concurrentes). Los resultados muestran un incremento progresivo del throughput (RPS) a medida que aumenta la concurrencia, lo que indica que el sistema posee capacidad de escalar horizontalmente y procesar un mayor número de solicitudes en condiciones de mayor demanda.

A pesar de este crecimiento en el throughput, la latencia presenta una característica clave: mientras el valor típico (p50) se mantiene extremadamente bajo en los tres escenarios (13–14 ms), el percentil alto (p95) permanece anclado en 5000 ms, coincidiendo con el valor del timeout configurado. Esta diferencia evidencia la presencia de un cuello de botella inherente al patrón síncrono REQ/REP, el cual forma una cola de espera cuando múltiples solicitudes compiten simultáneamente por una respuesta del Gestor de Carga.

Los fallos registrados no corresponden a errores funcionales del sistema distribuido ni a inconsistencias en la replicación; son errores HTTP 500 generados por el gateway HTTP–ZMQ cuando no obtiene respuesta dentro del tiempo límite. En otras palabras, el origen del error es la saturación del canal síncrono, no un fallo en la lógica de los actores, gestores o componentes de replicación.

En conjunto, los resultados evidencian un sistema capaz de procesar carga creciente, estable en condiciones normales y con comportamiento predecible bajo saturación, cuyo rendimiento se encuentra limitado principalmente por la naturaleza bloqueante de REQ/REP en escenarios de alta concurrencia.

4.3.3 Conclusión General Pruebas de Rendimiento

Las pruebas realizadas permiten caracterizar la respuesta del sistema distribuido bajo diferentes niveles de concurrencia:

- **El sistema escala adecuadamente**: a medida que aumentan los usuarios, también lo hace el RPS, demostrando que tanto los gestores como los actores pueden manejar mayor volumen de solicitudes sin degradarse inmediatamente.

- **La latencia típica (p50)** se mantiene en niveles muy bajos (13–14 ms), lo que confirma que la arquitectura es eficiente en condiciones normales.

- **El principal cuello de botella es el patrón REQ/REP**, pues bajo alta concurrencia las solicitudes quedan en cola, alcanzando el timeout de 5000 ms en el p95. Esto provoca la mayoría de errores observados en las pruebas.

- **Los errores HTTP 500 no provienen del sistema distribuido**, sino del gateway HTTP–ZMQ que expira al no recibir respuesta a tiempo. La arquitectura de backend (GC1–GC2–GA1–GA2–Actores) se mantuvo estable en todas las ejecuciones.

- **Las operaciones asíncronas (RENOVAR y DEVOLVER), basadas en PUB/SUB**, no generan picos de latencia porque no dependen del GC para una respuesta inmediata, lo que las hace más tolerantes a escenarios de carga elevada.

- **La replicación entre GA1 y GA2** continuó funcionando incluso durante el estrés, lo que demuestra la solidez del mecanismo de sincronización entre sedes.

Comparación de operaciones por patrón de comunicación

Operación	Patrón	Comportamiento observado	Observaciones técnicas
PRESTAR	REQ/REP	Alta latencia en p95, colas	Operación bloqueante en el GC; cuello de botella bajo concurrencia alta.
RENOVAR	PUB/SUB	Estable en todos los escenarios	No bloquea al GC; procesamiento asincrónico tolerante a picos de carga.
DEVOLVER	PUB/SUB	Estable en todos los escenarios	No depende de respuesta sincrónica; impacta poco en la latencia percibida por el usuario.

En conclusión, los resultados demuestran que la arquitectura distribuida implementada cumple con los criterios de funcionalidad, estabilidad y consistencia establecidos en el proyecto. El comportamiento del sistema bajo carga moderada es adecuado, y el análisis de percentiles confirma que el único factor limitante significativo es el procesamiento síncrono. Con mejoras orientadas a comunicación no bloqueante (async workers, timeouts adaptativos o colas internas), el sistema podría escalar de manera aún más eficiente ante niveles superiores de concurrencia.

5.Métricas de Desempeño

Para evaluar el rendimiento del Sistema Distribuido de Préstamo de Libros se definieron varias métricas fundamentales. Estas permiten analizar la eficiencia del procesamiento, la estabilidad bajo carga y la calidad del servicio. A continuación, se presenta cada métrica de forma clara y técnica.

1. Tiempo de respuesta (Response Time)

Mide cuánto tarda el sistema en procesar cada solicitud desde que el cliente la envía hasta que recibe la respuesta.

Incluye los siguientes valores:

- **Min:** tiempo más rápido observado
- **Average:** promedio de todos los tiempos
- **Max:** tiempo más lento observado
- **Percentiles:** 50, 90, 95 y 99

Importancia de los percentiles:

- **p50** → comportamiento normal del sistema
- **p95** → comportamiento bajo carga alta
- **p99** → picos y casos lentos
- **p100** → peor caso registrado

2. Requests Per Second (RPS)

Indica cuántas solicitudes puede procesar el sistema por segundo.
Es la métrica central para evaluar **capacidad de procesamiento** y **estabilidad bajo carga**.

3. Usuarios simultáneos

Representa la cantidad de clientes activos ejecutando solicitudes durante las pruebas.

Permite observar:

- La capacidad del sistema para **escalar horizontalmente**
- Si **mantiene el rendimiento** cuando aumenta la concurrencia

4. Cantidad total de solicitudes

Mide cuántas peticiones fueron procesadas por cada operación (PRESTAR, RENOVAR, DEVOLVER).

Sirve para validar:

- La **distribución real** de la carga
- El **volumen exacto** ejercido durante la prueba

5. Fallos (Fails)

Refleja el número de solicitudes que no pudieron ser procesadas correctamente.

Incluye:

- **# Fails:** fallos totales
- **Failures/s:** fallos por segundo

En un sistema distribuido estable, este valor **debe permanecer en cero**.

6. Uso de recursos del sistema (opcional si se monitorea externamente)

Permite identificar cuellos de botella internos.

Se pueden medir:

- Uso de **CPU**
- Consumo de **RAM**
- Actividad de **disco (I/O)**
- Uso de **red**
- Consumo de **SQLite** (locks, accesos, etc.)

7. Throughput

Cantidad de datos transmitidos por segundo.
Locust lo muestra como:

- **Average size (bytes) por request**

Es útil para identificar si el tamaño de las respuestas afecta el rendimiento.

8. Latencia

Mide el tiempo total de viaje del mensaje entre cliente y servidor.

Aunque se relaciona con el tiempo de respuesta, no son idénticos:

- **Latencia:** tiempo de red + espera
- **Response time:** latencia + procesamiento interno

9. Estabilidad del sistema

Refleja si el sistema mantiene un comportamiento uniforme bajo carga sostenida.

Se observa mediante:

- RPS estable
- Percentil 95 consistente
- Ausencia de errores o caídas
- No degradación del tiempo de respuesta durante la prueba

6.Conclusión

El desarrollo de este sistema distribuido permitió consolidar una arquitectura funcional, modular y tolerante a fallos, capaz de gestionar operaciones de préstamo, devolución y renovación entre dos sedes conectadas mediante ZeroMQ. A lo largo del proyecto se integraron modelos arquitecturales, de interacción y de fallos que sirvieron como base conceptual para la implementación, manteniendo coherencia entre el diseño y el comportamiento real del sistema.

La solución implementada no solo atiende los requisitos funcionales planteados, sino que también incorpora mecanismos robustos de replicación, supervisión y failover que garantizan continuidad operativa incluso ante fallos de componentes críticos. El uso de patrones como REQ/REP y PUB/SUB permitió separar correctamente los flujos sincrónicos y asincrónicos, mientras que la replicación GA1→GA2 aseguró consistencia eventual entre las sedes y redujo el riesgo de pérdida de información.

Las pruebas experimentales realizadas con diferentes niveles de carga (4, 6 y 10 procesos solicitantes por sede) evidenciaron la capacidad del sistema para manejar concurrencia, mantener tiempos de respuesta estables y recuperarse de fallos de forma transparente para el usuario. El análisis de métricas como latencia, throughput, RTO y RPO permitió validar empíricamente la resiliencia y el desempeño del sistema bajo escenarios exigentes.

En conjunto, el trabajo realizado demuestra una comprensión sólida de los conceptos fundamentales de los sistemas distribuidos, incluyendo comunicación entre procesos, asincronía, consistencia eventual, replicación de datos y detección de fallos. La implementación desarrollada constituye una base madura y bien estructurada que puede extenderse hacia nuevos servicios, un mayor número de sedes o mecanismos adicionales de optimización y escalabilidad.

El proyecto queda así preparado para su evaluación formal, mostrando una propuesta completa, coherente y técnicamente sustentada, que refleja el dominio progresivo de los elementos teóricos y prácticos abordados durante la asignatura.

En términos de escalabilidad, las operaciones implementadas con PUB/SUB demostraron un comportamiento más robusto bajo carga, manteniendo latencias estables al no depender de respuestas sincrónicas del Gestor de Carga. Asimismo, la replicación GA1→GA2 se mantuvo estable durante todos los experimentos, garantizando consistencia eventual y permitiendo que la sede secundaria asumiera el rol principal sin pérdida de información. Como limitación, el gateway HTTP–ZMQ basado en REQ/REP se identificó como un componente externo crítico que introduce tiempos de espera elevados y concentra gran parte de los timeouts observados, por lo que es un candidato natural a ser reemplazado por mecanismos asincrónicos o colas internas en futuras iteraciones.

7.Referencias

- Birman, K. (2012). *Guide to Reliable Distributed Systems: Building High-Assurance Applications and Cloud-Hosted Services.* Springer. https://doi.org/10.1007/978-1-4419-5705-8
- Coulouris, G., Dollimore, J., Kindberg, T., & Blair, G. (2013). *Distributed Systems: Concepts and Design* (5th ed.). Pearson Education Limited.
- Hintjens, P. (2013). *ZeroMQ: Messaging for Many Applications*. O'Reilly Media. Recuperado de https://zeromq.org
- Oracle Corporation. (2024). *MySQL 8.0 Reference Manual – Replication and High Availability.* Oracle. Recuperado de https://dev.mysql.com/doc/refman/8.0/en/replication.html
- Tanenbaum, A. S., & Van Steen, M. (2017). *Distributed Systems: Principles and Paradigms* (2nd ed.). Pearson / Vrije Universiteit Amsterdam.
- Prometheus Authors. (2024). *Prometheus Monitoring System – Documentation.* Recuperado de https://prometheus.io/docs
- Grafana Labs. (2024). *Grafana Documentation – Observability and Visualization Platform.* Recuperado de https://grafana.com/docs
- Hintjens, P. (2014). *Code Connected, Volume 1: Practical Messaging with ZeroMQ.* O'Reilly Media.
- Bharti, A. (2023, marzo). *PACELC Theorem Explained: A Comprehensive Guide to Distributed System Trade-offs.* Medium. Recuperado de https://medium.com/@amiteshbharti/pacelc-theorem-explained-a-comprehensive-guide-to-distributed-system-trade-offs-0f2a0748f3ba
- Python Software Foundation. (2024). *Python 3.12 Documentation – Multiprocessing and Concurrency.* Recuperado de https://docs.python.org/3/library/multiprocessing.html
- ZeroMQ Community. (2024). *ZGuide: The Guide to ØMQ – Patterns and Best Practices.* Recuperado de https://zguide.zeromq.org
- Fiebrink, R., & Gillies, M. (2018). *Measuring Performance and Reliability in Distributed Systems. ACM SIGCOMM Computer Communication Review, 48*(3), 35–41. https://doi.org/10.1145/3243157.3243165F
- SQLite Consortium. (2024). *Write-Ahead Logging (WAL) mode*. SQLite.org. https://www.sqlite.org/wal.html

- Locust Developers. (2024). *Locust: Scalable load testing framework—Documentation*. https://docs.locust.io
- Lynch, N. (1996). *Distributed algorithms*. Morgan Kaufmann. https://doi.org/10.1016/C2009-0-27660-5
- Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM, 21*(7), 558–565. https://doi.org/10.1145/359545.359563
- ZeroMQ Community. (2019). *ZMQ messaging patterns—RFC specifications*. https://rfc.zeromq.org/spec:37
- **Brewer, E. (2012).** CAP twelve years later: How the "rules" have changed. *Computer, 45*(2), 23–29. https://doi.org/10.1109/MC.2012.37
- **Gilbert, S., & Lynch, N. (2002).** Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News, 33*(2), 51–59. https://doi.org/10.1145/564585.564601
- **Vogels, W. (2009).** Eventually consistent. *Communications of the ACM, 52*(1), 40–44. https://doi.org/10.1145/1435417.1435432
- **Van Steen, M., & Tanenbaum, A. S. (2016).** *A concise introduction to distributed systems*. Maarten van Steen. https://www.distributed-systems.net
- **Hintjens, P. (2016).** *Scalability Protocols*. ZeroMQ RFC. https://rfc.zeromq.org/spec:16
- **Pike, R., Dorward, S., Griesemer, R., & Quinlan, S. (2015).** *The Text of Go concurrency patterns*. Google Research. https://talks.golang.org/2012/concurrency.slide
- **Jain, R. (1991).** *The Art of Computer Systems Performance Analysis*. John Wiley & Sons.
- **Barroso, L., Marty, M., Patterson, D., & Ranganathan, P. (2017).** Attack of the killer microseconds. *Communications of the ACM, 60*(4), 48–54. https://doi.org/10.1145/3015146
- **Schwaber, C. (2020).** Latency percentile analysis in distributed systems. *ACM Queue, 18*(5). https://queue.acm.org
- **Stonebraker, M., & Cattell, R. (2011).** 10 rules for scalable performance in "simple operation" datastores. *Communications of the ACM, 54*(6), 72–80. https://doi.org/10.1145/1953122.1953145
- **Patel, J. M. (2017).** Database replication: Concepts, algorithms, and trade-offs. *VLDB Tutorial*. https://www.vldb.org
- **Mauro, J. (2023).** Load testing microservices: Best practices using Locust and asynchronous gateways. *Medium Engineering*. https://medium.com
- **Microsoft Azure. (2024).** Load testing distributed workloads. https://learn.microsoft.com/azure/load-testing
- **Hohpe, G., & Woolf, B. (2003).** *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley.
- **Kreps, J. (2014).** *I Love Logs: Event Data, Stream Processing, and Data Integration*. O'Reilly Media.
-
-