

Proyecto Sistema Distribuido de Préstamo de Libros
Entrega 1.



Pontificia Universidad
JAVERIANA
Colombia

Daniel Avila Medina
Amelie Guerrero Jaramillo
Santiago Hernandez Barbosa
Andres Ortiz Forero

Sistemas Distribuidos

Osberth Cristhian Luef De Castro Cuevas
Pontificia Universidad Javeriana
Bogotá D.C

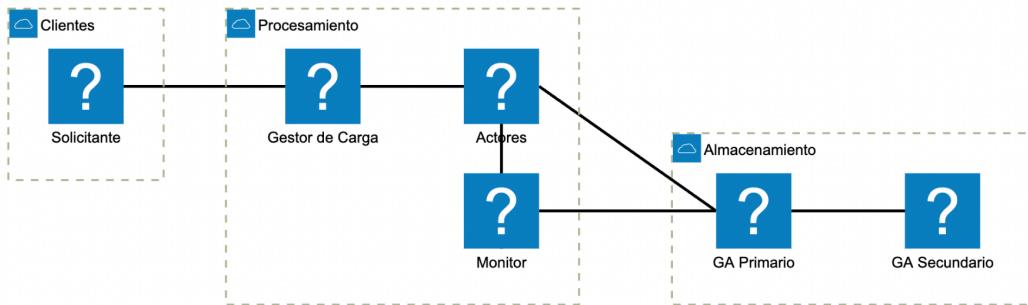
ÍNDICE

1. Modelos del Sistema
2. Diseño del Sistema
3. Protocolo de pruebas
4. Métricas de desempeño
5. Mecanismo para generar los requerimientos
6. Implementación Inicial

1. Modelos del sistema

Modelo Arquitectónico

El sistema se basa en una arquitectura distribuida compuesta por tres capas principales: la capa de clientes (PS) que genera solicitudes, la capa de procesamiento (GC y Actores) que gestiona y ejecuta las operaciones, y la capa de almacenamiento (GA) que mantiene la base de datos



Funciona de la siguiente manera nuestra implementación:

- Procesos Solicitantes (PS): leen archivos JSON línea a línea (≥ 20) y envían cada solicitud al GC usando ZeroMQ REQ.
- Gestor de Carga (GC): recibe por REP y responde inmediatamente {ok:true,msg:"Recibido y publicado"} al PS; pública el mismo mensaje por PUB a los tópicos DEVOLUCION y RENOVACION.
- Actores:
 - actor_devol (suscripto a tópico DEVOLUCION)
 - actor_renov (suscripto a tópico RENOVACION)Ambos envían al GA con REQ un apply_change según la operación.
- Gestor de Almacenamiento (GA): escucha REP, aplica cambios en SQLite (libros, prestamos, applied_ops), envía respuesta al Actor.
- BD SQLite: persistencia local; 1000 libros, ~200 préstamos activos (seed inicial).

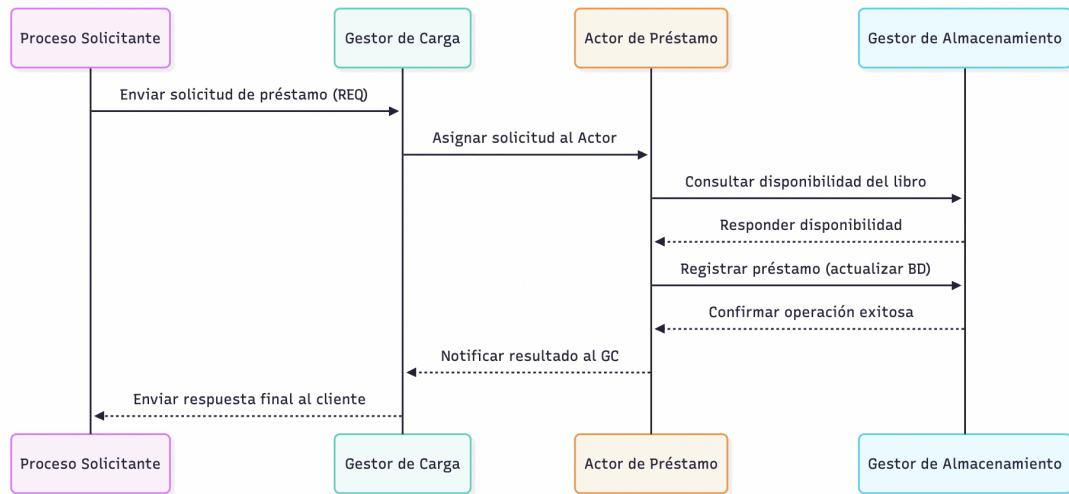
Conectores (patrones de integración):

- PS → GC: REQ/REP (petición/respuesta inmediata).
- GC → Actores: PUB/SUB (difusión por tópico).
- Actores → GA: REQ/REP (aplicar transacción y confirmar).

Atributos de calidad dirigidos por diseño:

- Desacople: PS no conoce a Actores/GA; GC desacopla.
- Escalabilidad: más PS se agregan sin tocar GC/Actores; más Actores por tipo si hiciera falta.
- Observabilidad: logs en cada salto; métricas propuestas (ver sección 3).
- Evolutividad: nuevos tópicos (p.ej., RESERVA) sin romper PS/GA actuales.

Modelo de Interacción



Funciona de la siguiente manera:

DEVOLUCIÓN

1. PS envía {op:"DEVOLUCION", idSolicitud, idUsuario, idLibro, sede, timestamp, idempotencyKey} → GC.REP.
2. GC responde inmediato {ok:true,"Recibido y publicado"} al PS y publica el mensaje por PUB en tópico DEVOLUCION.
3. actor_devol (SUB) recibe el payload y arma apply_change al GA.REP (REQ).
4. GA:
 - verifica idempotencia (applied_ops),
 - busca préstamo ACTIVO (libro, usuario, sede),
 - marca DEVUELTO y libera un ejemplar,
 - responde {ok:true,msg:...} al actor.

5. actor_devol loguea la confirmación.

RENOVACIÓN

1. PS → GC.REP (REQ) y recibe ack.
2. GC publica en RENOVACION.
3. actor_renov recibe; calcula nuevaFechaEntrega (+7 días si no viene), y hace REQ al GA.
4. GA valida préstamo ACTIVO y actualiza fecha_entrega, responde ok.
5. actor_renov loguea.

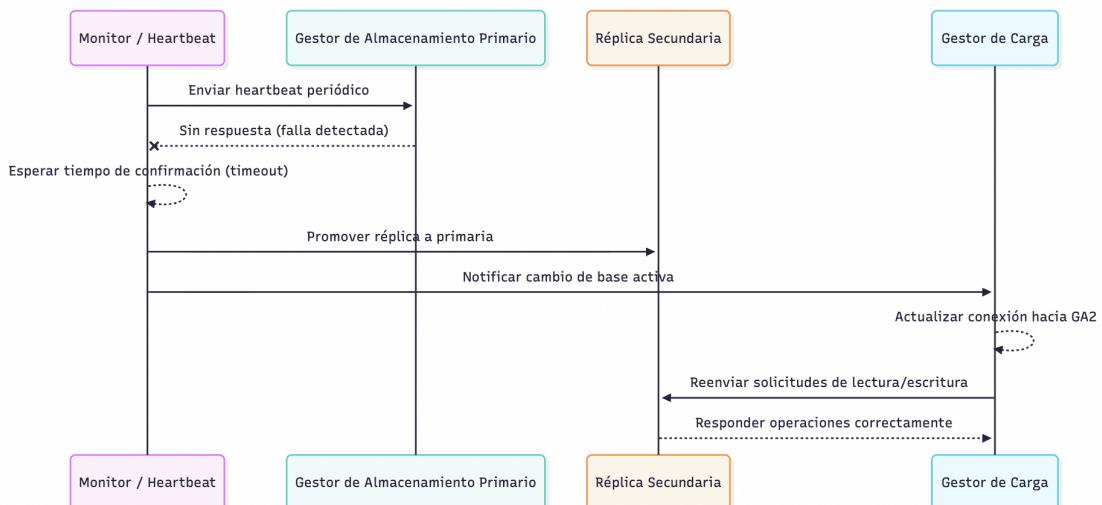
Modelo de Fallos

Tenemos que, para el momento, los riesgos y mitigaciones que tenemos son:

- PS ↔ GC: pérdida de conectividad → PS tiene timeout y reintentos (reconecta socket por intento).
- GC ↔ Actores (PUB/SUB): si el actor no está suscrito en el momento de la publicación, ZeroMQ no retiene; se puede perder esa publicación.
 - Mitigación parcial: arrancar primero Actores y después GC y PS.
 - Idempotencia en GA evita efectos duplicados cuando haya reintentos.
- Actores ↔ GA: timeout o caída de GA → REQ con timeout y reintento manual (para Entrega 1 dejamos 1 intento; en la final podemos ampliar).
- Duplicados: si PS reintenta o el actor reintenta, GA usa applied_ops(idempotencyKey) para volver idempotentes las operaciones.
- Consistencia: GA usa transacciones ACID (SQLite); por lo que cada operación es atómica.

Componentes de “enmascaramiento” planificados para la entrega final (quedan descritos en el informe de diseño):

- Persistencia de publicaciones en GC (cola local con reenvío hasta confirmación de actor).
- Health-check (La idea es que funcione tal cual como lo hacía en el taller 1 y 2, con un failover básico para GA (réplica pasiva + heartbeat y conmutación manual/semitáctica)).
- Reintentos exponenciales en Actores hacia GA con backoff/jitter.



Modelo de Seguridad

Nos encontramos con que la superficie de ataque son los puertos 5555 (GC-REP), 5560 (GC-PUB), 5570 (GA-REP), donde las principales amenazas son el spoofing de mensajes, inyección SQL, fuga de datos.

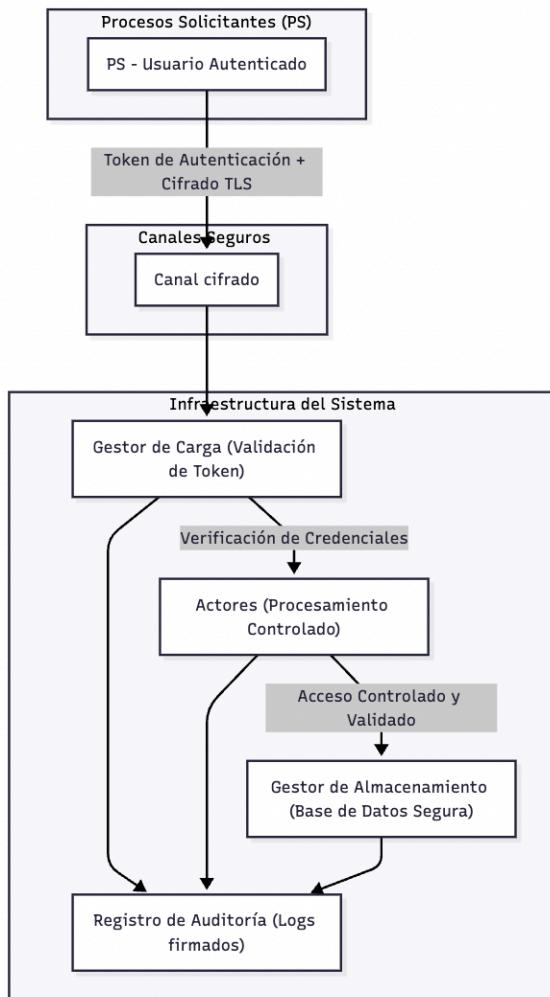
Controles aplicados a este momento:

- Validación de contratos JSON en PS y GA (campos obligatorios).
- Consultas parametrizadas en SQLite (sin concatenar SQL).
- Segmentación de red (correr en LAN/VM bridged; puertos abiertos sólo a subred de laboratorio).
- Trazabilidad básica (logs con idSolicitud e idempotencyKey).

Plan para versión final:

- Cifrado/TLS a nivel de túnel (stunnel/SSH Port Forward) o CurveZMQ.
- Autenticación ligera por token compartido en el contrato (valido en GC/GA).

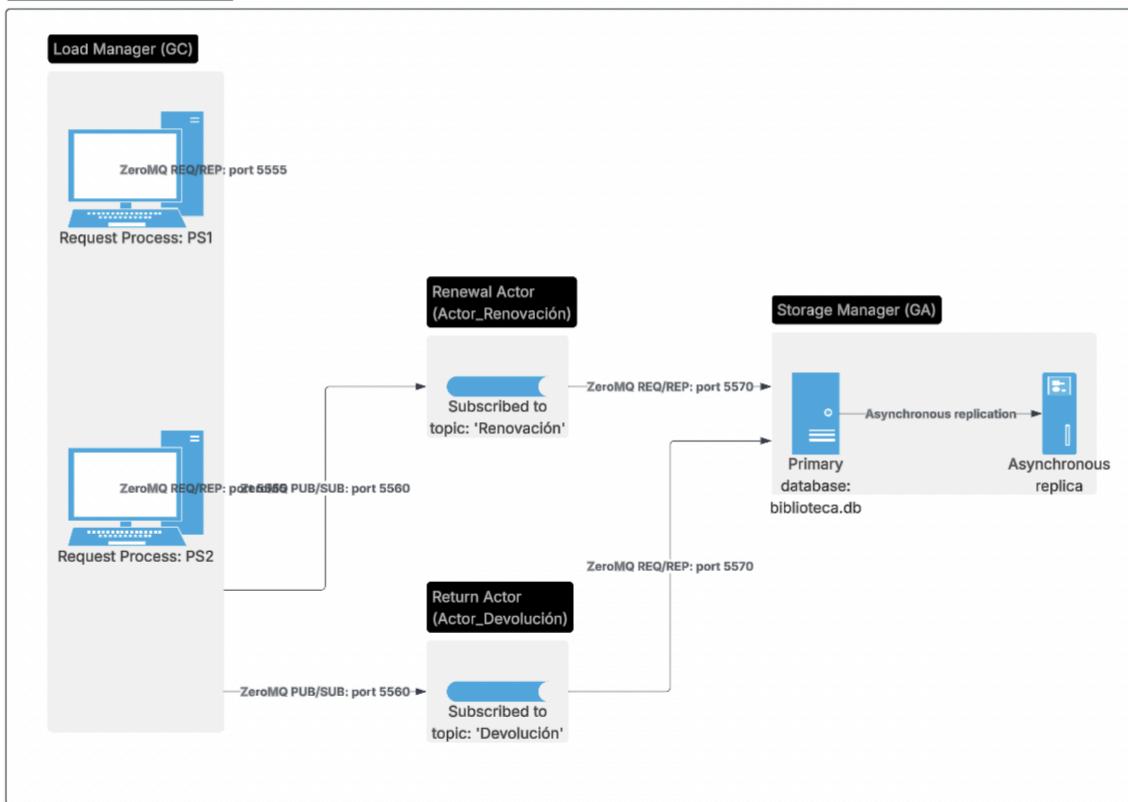
- Sanitización extra y límites de tasa (rate limiting) en GC.



2. Diseño del sistema

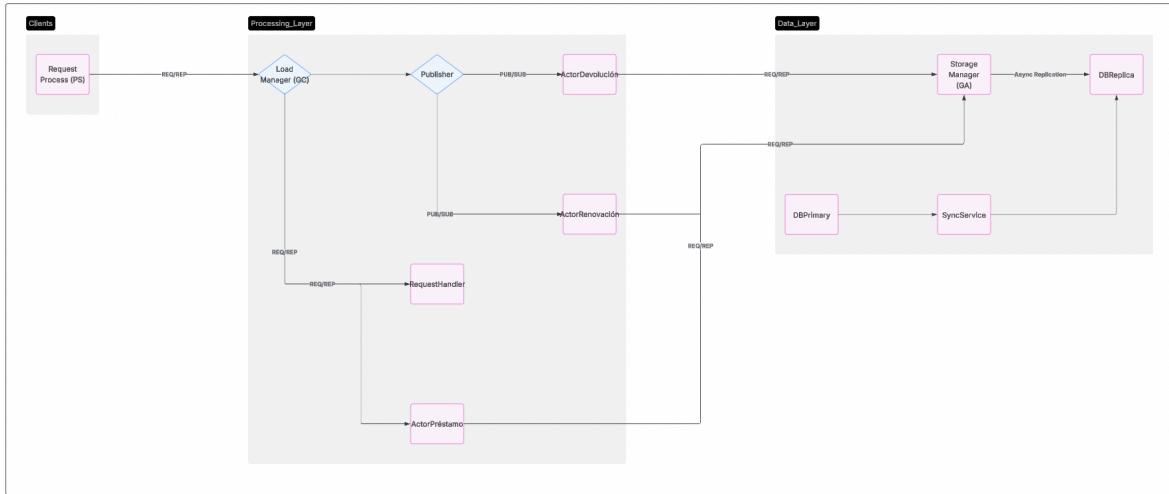
Diagrama de despliegue

Distributed Library System Infrastructure



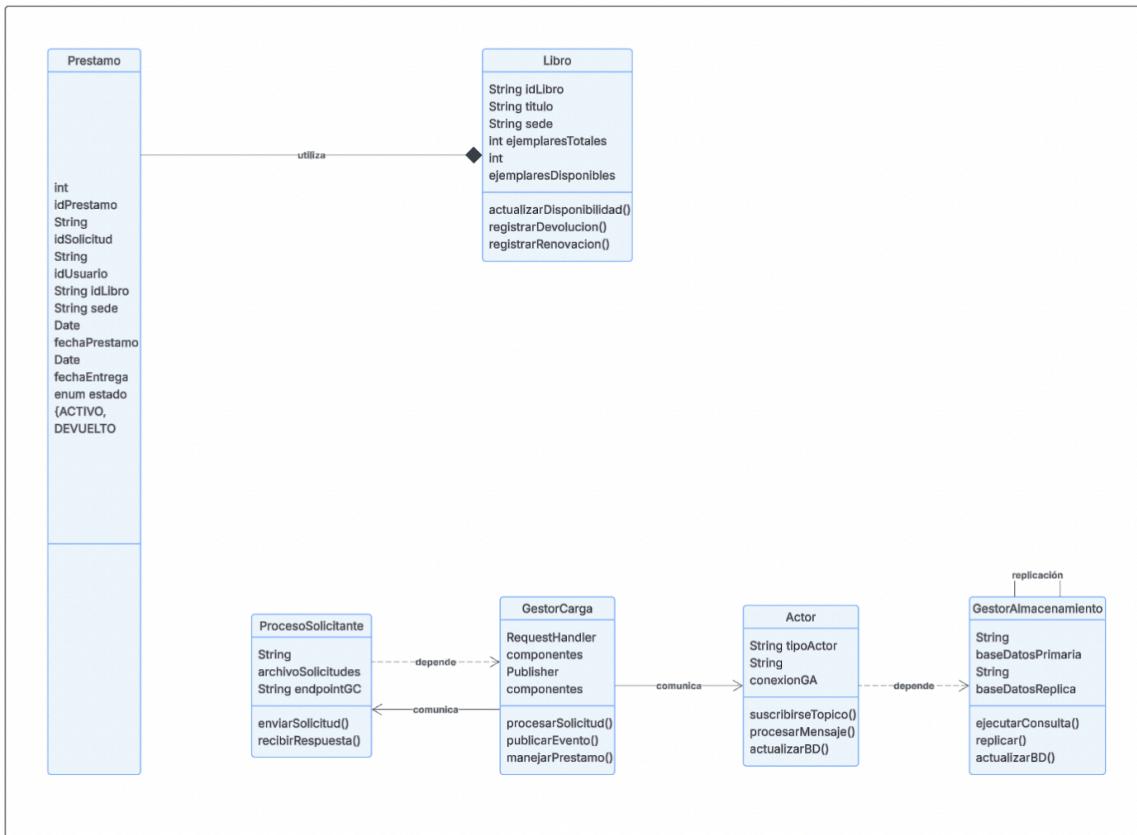
Este diagrama muestra cómo se distribuyen físicamente los procesos del sistema en diferentes máquinas o VMs. Cada nodo (GA, GC, Actores y PS) se comunica mediante ZeroMQ usando distintos patrones: REQ/REP para solicitudes síncronas (como préstamos) y PUB/SUB para mensajes asíncronos (como devoluciones y renovaciones). El Gestor de Almacenamiento (GA) mantiene una base de datos principal y una réplica que se actualiza de manera asíncrona, garantizando disponibilidad ante fallas. Así, el diagrama refleja la infraestructura distribuida y la interconexión entre sedes.

Diagrama de componentes



El diagrama de componentes describe la arquitectura lógica del sistema, mostrando cómo cada módulo se relaciona e intercambia información. Los Procesos Solicitantes (PS) generan peticiones de los usuarios hacia el Gestor de Carga (GC), que actúa como intermediario. El GC envía los mensajes a los Actores, quienes procesan las devoluciones, renovaciones o préstamos y actualizan la base de datos a través del Gestor de Almacenamiento (GA). Cada componente tiene una función definida y se comunica con otros mediante mensajes ZeroMQ, lo que facilita el desacoplamiento y la escalabilidad del sistema.

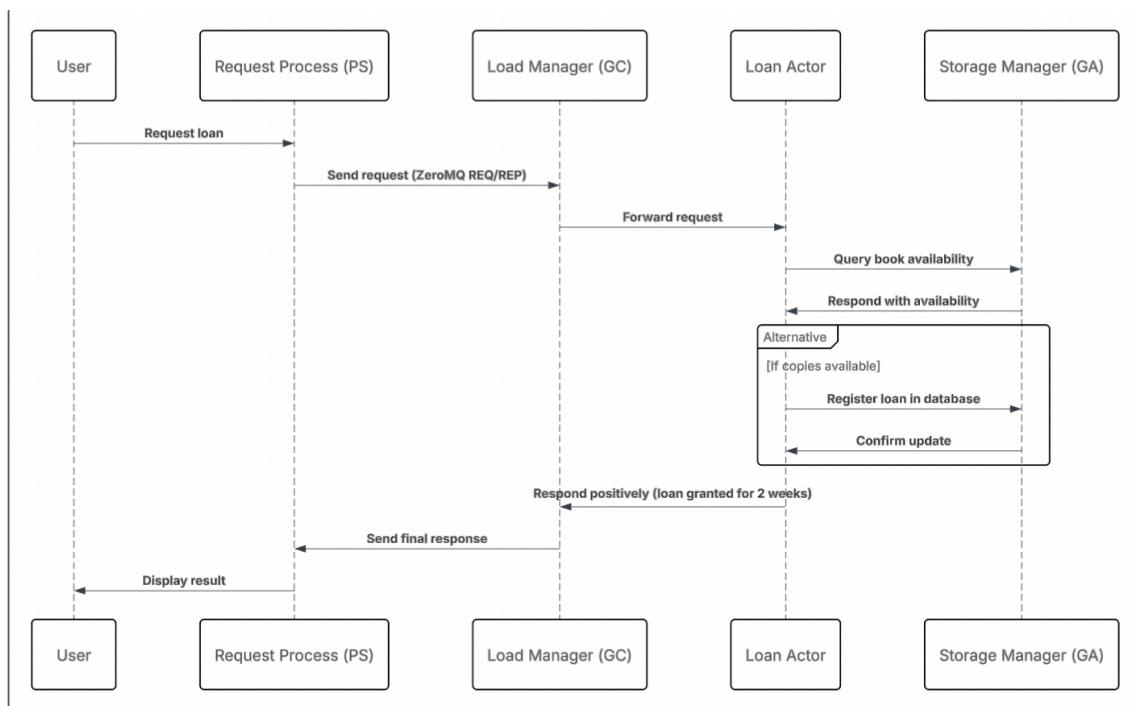
Diagrama de clases



El diagrama de clases modela la estructura orientada a objetos del sistema. Las clases principales —Libro, Préstamo, ProcesoSolicitante, GestorCarga, Actor y GestorAlmacenamiento— representan las entidades y controladores del sistema. Las relaciones entre ellas reflejan la lógica de negocio: un préstamo está asociado a un libro, los actores dependen del gestor de almacenamiento para modificar datos, y el gestor de carga coordina la comunicación entre clientes y procesos internos. Este diagrama sirve para entender la organización del código y las responsabilidades de cada clase.

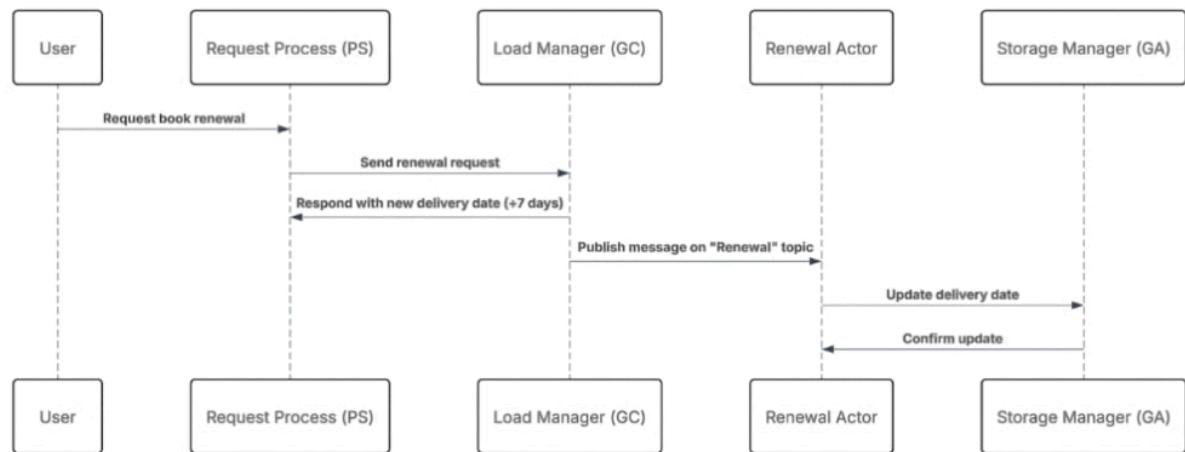
Diagramas de secuencia

Diagrama de Secuencia: Solicitar Préstamo



En este flujo, el usuario solicita un libro a través del Proceso Solicitante (PS), que envía la petición al Gestor de Carga (GC) mediante una comunicación síncrona (ZeroMQ REQ/REP). El GC reenvía la solicitud al Actor de Préstamo, quien consulta con el Gestor de Almacenamiento (GA) si hay ejemplares disponibles en la base de datos. Si el GA confirma disponibilidad, el actor registra el préstamo y devuelve una respuesta positiva al GC, que a su vez responde al PS. Finalmente, el PS muestra al usuario la confirmación del préstamo (por 2 semanas). Este flujo refleja una transacción completa y bloqueante, donde el usuario espera la respuesta del sistema antes de continuar.

Diagrama de Secuencia: Renovar préstamo



En este caso, el usuario solicita la renovación de un libro ya prestado mediante el PS, que envía la solicitud al GC. El Gestor de Carga responde de inmediato al PS con la nueva fecha de entrega (una semana adicional), sin esperar actualización en la base de datos. Luego, el GC publica un mensaje asíncrono en el tópico “Renovación” usando ZeroMQ PUB/SUB. El Actor de Renovación, suscrito a este tópico, recibe el mensaje y actualiza la fecha en el Gestor de Almacenamiento (GA), quien confirma la modificación. Este flujo representa una operación asíncrona, donde la respuesta al usuario es inmediata, pero la actualización en la BD ocurre en segundo plano.

3. Protocolo de pruebas

Pruebas funcionales (happy path y borde)

- F1 – Flujo DEVOLUCIÓN: 10 líneas sede1 → ver ack PS, publicación en GC, recepción en actor_devol, confirmación GA y cambio en BD (prestamos.estado=DEVUELTO, libros.ejemplares_disponibles +1).
- F2 – Flujo RENOVACIÓN: 10 líneas sede2 → ver ack PS, publicación en GC, recepción en actor_renov, confirmación GA y cambio de fecha_entrega.
- F3 – Contrato inválido: línea con op desconocida → GC responde {ok:false} y no publica.
- F4 – Idempotencia GA: repetir dos veces la misma solicitud (mismo idempotencyKey) → GA responde “Ya aplicado” y no duplica efectos.
- F5 – Falta de préstamo activo (DEVOL/RENOV): GA responde {ok:true,msg:"No había préstamo activo"} o {ok:false, msg:"No hay préstamo activo para renovar"}.

Pruebas de robustez y fallas

- R1 – Actores arrancan después de GC: demostrar que las primeras publicaciones pueden perderse; dejar constancia (motivación para cola/confirmación en la final).

- R2 – GC caído: ejecutar PS con --endpoint hacia GC apagado → ver timeout y reintentos.
- R3 – GA caído: levantar actores con GA apagado → ver timeout en actores al pedir a GA; luego levantar GA y reintentar una operación.
- R4 – Latencia en GA: insertar sleep artificial (si se decide) y verificar que Actores respetan timeout.
- R5 – Concurrencia: correr sede1 y sede2 al tiempo (dos PS) y verificar que GA mantiene atomicidad.

Pruebas de desempeño (éxito y estrés)

- P1 – Throughput base: 1 PS (archivo 20 ops) → medir ops/s end-to-end (PS ack, Actor→GA ok).
- P2 – Throughput multi-sede: 2 PS simultáneos (sede1+sede2) → medir ops/s y latencias P50/P95.
- P3 – Escalado PS: duplicar tamaño de archivo (40/80/160 ops) y observar linealidad.
- P4 – Límite: correr 3–4 PS en paralelo (copias del mismo archivo) y observar dónde aparece cuello de botella (CPU/IO/locks).

Criterios de aceptación

- Ack inmediato del GC (< 50 ms en LAN).
- Respuesta GA por operación < 150 ms (promedio) en localhost; en VM, < 300 ms.
- Sin errores en BD y consistencia mantenida.

4. Métricas de desempeño

Métricas a recolectar:

Latencias:

- t_ack (PS): tiempo desde send → respuesta GC (ack inmediato).
- t_ga (Actor): tiempo desde recepción del tópico → respuesta GA.

Throughput:

- ops/s en PS (por archivo y total).
- ops/s aplicadas en GA (contadas por minuto).

Recursos:

- CPU y RAM de GC y GA.
- I/O de disco de GA (SQLite).
- Tráfico de red (bytes TX/RX por proceso o interfaz).

Ideas que tenemos para obtener estas métricas es mediante instrumentación en el código

- PS: medir t0=time.monotonic() antes de send_json, t1 después de recv_json. Log a CSV:
ps_metrics.csv: op,idSolicitud,t_ack_ms,timestamp
- Actores: t0 al recibir tópico, t1 al recibir respuesta GA.
actor_metrics.csv: actor,op,idSolicitud,t_ga_ms,timestamp
- GA: medir duración de cada transacción (BEGIN→COMMIT).
ga_metrics.csv: op,idSolicitud,tx_ms,timestamp

Para luego calcular P50/P95/P99 con Excel o Pandas

5. Mecanismo para generar los requerimientos

Nuestro mecanismo de generación de requerimientos consiste en leer, desde los Procesos Solicitantes (PS), archivos de texto con una solicitud JSON por línea (formato JSONL). Cada línea representa una operación del sistema (DEVOLUCION o RENOVACION) que se envía al Gestor de Carga (GC) mediante ZeroMQ (patrón REQ/REP). Este enfoque cumple explícitamente el requisito del enunciado: “Mecanismo para generar los requerimientos (lectura de un archivo o usando un generador de carga)”, usando la opción de lectura de archivos.

5.1 Estructura de carpetas y archivos de entrada

Los archivos de solicitudes se ubican en:

- ps/data/sol_sede1.txt
- ps/data/sol_sede2.txt

Cada archivo contiene ≥10 líneas (al menos 20 en total). Cada línea es un JSON independiente (una solicitud). Ejemplos reales incluidos:

```
{"op":"DEVOLUCION","idSolicitud":"S-001","idUsuario":"U001","idLibro":"L100"
,"sede":"SEDE1","timestamp":"2025-10-07T21:00:00Z"}
{"op":"RENOVACION","idSolicitud":"S-002","idUsuario":"U002","idLibro":"L101"
,"sede":"SEDE1","timestamp":"2025-10-07T21:00:05Z"}
```

...

y para SEDE2:

```
{"op":"RENOVACION","idSolicitud":"S-011","idUsuario":"U011","idLibro":"L110",
"sede":"SEDE2","timestamp":"2025-10-07T21:01:00Z"}
 {"op":"DEVOLUCION","idSolicitud":"S-012","idUsuario":"U012","idLibro":"L111"
,"sede":"SEDE2","timestamp":"2025-10-07T21:01:05Z"}
```

...

5.2 Contrato del mensaje (campos y semántica)

Cada línea JSON debe incluir:

- op: “DEVOLUCION” o “RENOVACION”.
- idSolicitud: identificador único de la solicitud (si falta, el PS lo autogenera).
- idUsuario: usuario solicitante (obligatorio).
- idLibro: identificador del libro (obligatorio).
- sede: “SEDE1” o “SEDE2” (obligatorio).
- timestamp: ISO-8601 en UTC (si falta, el PS lo completa con la hora actual).
- idempotencyKey: clave para idempotencia en GA (si falta, el PS la deriva de op:idSolicitud:idLibro).

5.3 Cómo se procesan las líneas (flujo PS)

- ps/ps.py abre el archivo indicado con --file.
- Lee el archivo línea a línea; ignora líneas vacías.
- Parsea cada línea a dict y valida el contrato (op, campos obligatorios).
- Completa valores faltantes: timestamp e idempotencyKey.
- Crea un socket REQ hacia el GC (--endpoint) y envía el JSON.
- Espera la respuesta inmediata del GC (ack).
- Registra en consola el resultado y pasa a la siguiente línea.

Este esquema desacopla la generación de carga (archivo) del transporte; además evita bloquear al PS por largos tiempos porque el GC responde inmediatamente y la aplicación real se completa asíncronamente vía Actores y GA.

5.4 Parámetros de envío y tolerancia a fallos en PS

--file: ruta al archivo JSONL (obligatorio).
--endpoint: dirección del GC REP (por defecto tcp://127.0.0.1:5555 en localhost; en VMs se cambia a la IP del GC).
--interval: intervalo entre envíos (segundos), p. ej. 0.2 para espaciar solicitudes.
--timeout_ms: tiempo máximo de espera del ack del GC (por defecto 2000 ms).
--retries: reintentos por solicitud (para Entrega 1 mantenemos 1 intento; puede ampliarse en la entrega final).

Si se produce un timeout (GC no responde), el PS lo informa por consola y puede reintentar según --retries. Cada envío usa un socket REQ nuevo para simplificar la reconexión limpia.

5.5 Ejecución (comandos)

Localhost (con GC escuchando en 5555):

- python -m ps.ps --file ps/data/sol_sede1.txt --endpoint tcp://127.0.0.1:5555
- python -m ps.ps --file ps/data/sol_sede2.txt --endpoint tcp://127.0.0.1:5555

En VMs (PS apuntando a la IP real del GC):

- python -m ps.ps --file ps/data/sol_sede1.txt --endpoint tcp://<IP_GC>:5555
- python -m ps.ps --file ps/data/sol_sede2.txt --endpoint tcp://<IP_GC>:5555

5.6 Manejo de errores y trazabilidad

- Líneas inválidas (JSON mal formado o campos faltantes): el PS las registra como [ERROR] y continúa con la siguiente línea, para no frenar el lote.
- Respuesta no-JSON del GC o timeouts: se muestran como [WARN]/[ERROR] y, si hay reintentos habilitados, se reintenta.
- Trazabilidad: cada log del PS incluye op e idSolicitud. En GA, además, se registra la idempotencyKey aplicada para evitar efectos duplicados.

6. Implementación Inicial

Para comprobar el funcionamiento extremo a extremo levantamos seis terminales en este orden.

1. Terminal 1 (GA): python -m ga.ga --rep tcp://*:5570 --db ga/biblioteca.db inicia el Gestor de Almacenamiento escuchando en 5570 y operando sobre la base SQLite.
2. Terminal 2 (Actor DEVOL): python -m actores.actor_devol --sub tcp://127.0.0.1:5560 --ga tcp://127.0.0.1:5570 se suscribe al tópico DEVOLUCION del GC y conecta por REQ al GA.
3. Terminal 3 (Actor RENOV): python -m actores.actor_renov --sub tcp://127.0.0.1:5560 --ga tcp://127.0.0.1:5570 se suscribe al tópico RENOVACION y también conecta al GA.
4. Terminal 4 (GC): python -m gestor_carga.gc --rep tcp://*:5555 --pub tcp://*:5560 arranca el Gestor de Carga con REP para PS (5555) y PUB para actores (5560).
5. Terminal 5 (PS Sede 1): python -m ps.ps --file ps/data/sol_sede1.txt --endpoint tcp://127.0.0.1:5555 lee el archivo JSONL de SEDE1 y envía cada línea al GC (REQ/REP). El PS muestra el ack inmediato del GC {ok: true, msg: "Recibido y publicado"} por solicitud.
6. Terminal 6 (PS Sede 2): python -m ps.ps --file ps/data/sol_sede2.txt --endpoint tcp://127.0.0.1:5555 hace lo propio para SEDE2.

Con este flujo se observa: PS recibe acks inmediatos; GC registra “Recibí / Publicado”; cada actor recibe su tópico (PUB/SUB) y solicita al GA aplicar el cambio (REQ/REP); el GA confirma operación y persiste en SQLite. Al re-ejecutar los PS con las mismas solicitudes, el

GA detecta la idempotencia vía idempotencyKey y responde “ya aplicado (idempotente)”, evitando efectos duplicados.

Flujo correcto:

The image displays four terminal windows illustrating the correct flow of operations between actors and servers. The windows are arranged in a 2x2 grid. The top row shows interactions between the 'actor' and 'actor_devol' components, while the bottom row shows interactions between the 'gestor_carga_gc' and 'actor_devol' components. The left column shows the actor's perspective, and the right column shows the server's perspective. The logs show a sequence of operations: RENOVACION (with idempotencyKey), DEVOLUCION (with idempotencyKey), and another RENOVACION (with idempotencyKey). The logs also include timestamps and status codes (e.g., 2025-10-07T21:01:15Z, 2025-10-07T21:01:16Z, etc.). The right-hand logs provide detailed information about each operation, including the received idempotencyKey and the corresponding response from the server.

```

C:\Users\santi\Downloads\Proyecto_Distribuidos> python -m ga.ga --rep tcp://:5570 --db ga/biblioteca
[...]
C:\Users\santi\Downloads\Proyecto_Distribuidos> python -m actores.actor_devol --sub tcp://:127.0.0.1:5569
[...]
C:\Users\santi\Downloads\Proyecto_Distribuidos> python -m actores.actor_devol --sub tcp://:127.0.0.1:5570
[...]
C:\Users\santi\Downloads\Proyecto_Distribuidos> python -m gestor_carga_gc --rep tcp://:5555 --pub tcp://:5556
[...]
C:\Users\santi\Downloads\Proyecto_Distribuidos> python -m actores.actor_renov --sub tcp://:127.0.0.1:5569
[...]
C:\Users\santi\Downloads\Proyecto_Distribuidos> python -m actores.actor_renov --sub tcp://:127.0.0.1:5570
[...]
C:\Users\santi\Downloads\Proyecto_Distribuidos> python -m powershell -File ps\data\sol_sede1.txt --endpoint tcp://127.0.0.1:5555
[...]
C:\Users\santi\Downloads\Proyecto_Distribuidos> python -m powershell -File ps\data\sol_sede2.txt --endpoint tcp://127.0.0.1:5555
[...]

```

Idempotencia:

```
[PS] Terminado. total=10 ok=10 fail=0
(.venv) PS C:\Users\santi\Downloads\Proyecto_Distribuidos\Proyecto_Distribuidos> python -m p
● s.ps --file ps/data/sol_sede1.txt --endpoint tcp://127.0.0.1:5555
[PS] Enviando solicitudes a tcp://127.0.0.1:5555
[PS][OK] DEVOLUCION id=S-001 → {'ok': True, 'msg': 'Recibido y publicado'}
[PS][OK] RENOVACION id=S-002 → {'ok': True, 'msg': 'Recibido y publicado'}
[PS][OK] RENOVACION id=S-003 → {'ok': True, 'msg': 'Recibido y publicado'}
[PS][OK] DEVOLUCION id=S-004 → {'ok': True, 'msg': 'Recibido y publicado'}
[PS][OK] DEVOLUCION id=S-005 → {'ok': True, 'msg': 'Recibido y publicado'}
[PS][OK] RENOVACION id=S-006 → {'ok': True, 'msg': 'Recibido y publicado'}
[PS][OK] DEVOLUCION id=S-007 → {'ok': True, 'msg': 'Recibido y publicado'}
[PS][OK] RENOVACION id=S-008 → {'ok': True, 'msg': 'Recibido y publicado'}
[PS][OK] RENOVACION id=S-009 → {'ok': True, 'msg': 'Recibido y publicado'}
[PS][OK] DEVOLUCION id=S-010 → {'ok': True, 'msg': 'Recibido y publicado'}
[PS] Terminado. total=10 ok=10 fail=0
❖ (.venv) PS C:\Users\santi\Downloads\Proyecto_Distribuidos\Proyecto_Distribuidos>
```

```
[GA] DEVOLUCION id=S-001 → ya aplicado (idempotente)
[GA] RENOVACION id=S-002 → ya aplicado (idempotente)
[GA] RENOVACION id=S-003 → ya aplicado (idempotente)
[GA] DEVOLUCION id=S-004 → ya aplicado (idempotente)
[GA] DEVOLUCION id=S-005 → ya aplicado (idempotente)
[GA] RENOVACION id=S-006 → ya aplicado (idempotente)
[GA] DEVOLUCION id=S-007 → ya aplicado (idempotente)
[GA] RENOVACION id=S-008 → ya aplicado (idempotente)
[GA] RENOVACION id=S-009 → ya aplicado (idempotente)
[GA] DEVOLUCION id=S-010 → ya aplicado (idempotente)
```

Verificando el correcto funcionamiento de esta versión inicial de nuestra implementación

Bibliografía:

1. Hintjens, P., & comunidad ØMQ. (s. f.). *ØMQ – The Guide*. ZeroMQ. <https://zguide.zeromq.org/docs/> (zguide.zeromq.org)
2. ZeroMQ Project. (s. f.). *Socket API*. ZeroMQ. <https://zeromq.org/socket-api/> (zeromq.org)
3. PyZMQ Developers. (s. f.). *PyZMQ Documentation*. Read the Docs. <https://pyzmq.readthedocs.io/> (pyzmq.readthedocs.io)
4. Python Software Foundation. (s. f.). *sqlite3 — DB-API 2.0 interface for SQLite databases*. Python Documentation. <https://docs.python.org/3/library/sqlite3.html> ([Python documentation](https://docs.python.org/3/library/sqlite3.html))