Facultad de Ingeniería
Departamento de Ingeniería de Sistemas

Introducción a Sistemas Distribuidos

TALLER ZeroMQ

Sistema Distribuido para Gestión de Biblioteca usando Cliente - Servidor
Video explicativo

Integrantes:
Ana Sofía Arboleda
Daniel Felipe Ramírez Vargas
Guillermo Andrés Aponte Cárdenas
Samuel Pico Durán

Docente: John Jairo Corredor

Fecha: 27 de febrero de 2026

Índice general

Capítulo 1

Introducción

1.1. Contexto del Problema

El presente trabajo fue desarrollado en la sala Alan Turing, utilizando tres computadores conectados a la misma red LAN. La arquitectura implementada consistió en un modelo cliente-servidor, donde uno de los equipos actuó como servidor y los otros dos como clientes.

La interacción del usuario se realizó mediante una aplicación web ejecutada en entorno local. A través de esta interfaz, los clientes envían peticiones HTTP (GET y POST), las cuales son procesadas por una aplicación desarrollada en Python utilizando Flask. Posteriormente, dichas solicitudes eran transformadas y enviadas al servidor mediante el la librería de comunicación ZeroMQ, bajo un modelo de comunicación síncrono.

El servidor, encargado de recibir las peticiones, procesa cada solicitud realizando las operaciones correspondientes sobre una base de datos almacenada en formato JSON. Dependiendo del tipo de operación (préstamo, consulta o devolución), el servidor actualiza la base de datos cuando es necesario y envía una respuesta al cliente. Finalmente, el servidor continúa en estado de escucha, preparado para atender nuevas solicitudes.

1.2. Justificación

Aunque la especificación del taller no exigía el desarrollo de una interfaz web, se decidió implementar una aplicación basada en Flask debido a que facilita la programación en Python, hace el sistema más intuitivo y permite visualizar y probar rápidamente cada funcionalidad.

El uso de una interfaz web permitió realizar pruebas más claras y dinámicas, facilitando la validación de cada método remoto y la verificación de los cambios en la base de datos. Además, esta decisión hizo el sistema más organizado y sencillo de evaluar visualmente.

Adicionalmente, el taller se desarrolló con el propósito de fortalecer el aprendizaje sobre la comunicación distribuida mediante ZeroMQ, aplicando la arquitectura cliente-servidor en una red LAN real. Esta experiencia permitió comprender de manera práctica el flujo completo de las peticiones: desde la generación de una solicitud HTTP, su transformación en un mensaje ZeroMQ, el procesamiento en el servidor y la posterior respuesta al cliente.

Capítulo 2

Objetivos

2.1. Objetivo General

Implementar un sistema de gestión de biblioteca distribuido utilizando ZeroMQ, aplicando los principios fundamentales de sistemas distribuidos como la comunicación cliente-servidor, la concurrencia y el control de acceso a recursos compartidos.

2.2. Objetivos Específicos

- Construir un servidor capaz de atender varias solicitudes al mismo tiempo, sin que una petición tenga que esperar a que terminen todas las anteriores.
- Desarrollar una interfaz que permita a los usuarios realizar operaciones sobre el catálogo de la biblioteca (préstamos, consultas y devoluciones) desde cualquier computador conectado a la red.
- Definir un formato claro de comunicación entre el cliente y el servidor, de modo que ambos se entiendan sin importar en qué máquina estén corriendo.
- Garantizar que varios usuarios hagan cambios al mismo tiempo y los datos no se queden en un estado inconsistente.

Capítulo 3

Marco Teórico

3.1. Servicio Remoto

Un servicio remoto es cualquier programa, función o recurso informático que se puede encontrar alojado en un servidor o dispositivo distinto al que el usuario está utilizando físicamente (el cliente), y al cual se accede a través de una red. La forma más conocida de este paradigma es la Llamada a Procedimiento Remoto (RPC), que permite ejecutar una subrutina en otro espacio de direcciones como si fuera local. En este sistema, la invocación remota se implementa sobre ZeroMQ con serialización JSON, abstrayendo completamente la red a través de la clase LibraryClient como se observa en la figura.

3.2. ZeroMQ

ZeroMQ (ZMQ) es una biblioteca de mensajería asíncrona entre aplicaciones. ZMQ maneja automáticamente la reconexión, el encolado de mensajes y el balanceo de carga, simplificando el desarrollo de aplicaciones distribuidas. ZeroMQ ofrece patrones de comunicación como **REQ-REP** (Request-Reply) que es el más simple: el cliente envía una solicitud y espera una respuesta. El patrón **ROUTER-DEALER** es una extensión del anterior que permite distribuir las solicitudes entre múltiples workers de forma asíncrona. También existen patrones más complejos como **PUB-SUB** (Publish-Subscribe) que conecta publicadores que emiten datos con suscriptores que reciben solo aquellos en los que están interesados.

3.3. Arquitectura Cliente-Servidor

La arquitectura del sistema separa las responsabilidades en dos capas: el servidor ZeroMQ (lógica de negocio y datos) y el cliente web (interfaz de usuario). El cliente no accede directamente a la base de datos; toda operación debe pasar por el servidor ZMQ. Esta separación facilita el mantenimiento, permite escalar cada capa independientemente y mejora la seguridad al centralizar el control de acceso a los datos.

3.4. Patrón REQ-REP (Request-Repy)

El patrón REQ-REP funciona con sockets(son por donde un programa manda o recibe datos a través de la red).

- El **request** es el que pregunta. En el proyecto es LibraryClient. Manda un mensaje y se queda esperando bloqueado hasta que llegue la respuesta.
- El **reply** es el que responde. En el proyecto es el servidor **(library_service .py)**. Espera un mensaje, lo procesa, responde, y solo entonces puede recibir el siguiente.

El ciclo siempre es enviar, recibir, enviar, recibir.

3.5. Flask y HTMX

Flask es una herramienta para construir páginas web con Python, su trabajo es recibir las peticiones HTTP del cliente, procesarlas y devolver una respuesta. En este proyecto, Flask recibe las acciones del usuario (prestar el libro con ISBN, prestar libro por título...) y las transmite al servidor de biblioteca. HTMX es una librería JavaScript que hace que las páginas web se actualicen sin recargar todo el navegador. Cuando el usuario hace clic en 'Prestar', HTMX manda la solicitud en segundo plano y dibuja solo el recuadro de resultado con la nueva información.

Capítulo 4

Diseño del Sistema

En esta sección se describe cómo fue pensado y estructurado el sistema antes de escribir código: qué partes lo componen, cómo se comunican entre sí, qué decisiones de diseño se tomaron y por qué.

4.1. Arquitectura General

El sistema implementa una arquitectura distribuida tipo cliente-servidor, compuesta por tres capas principales:

1. la interfaz de usuario (navegador web),
2. el cliente web (Flask + LibraryClient),
3. el servidor de biblioteca (ZeroMQ REP + DB.json).

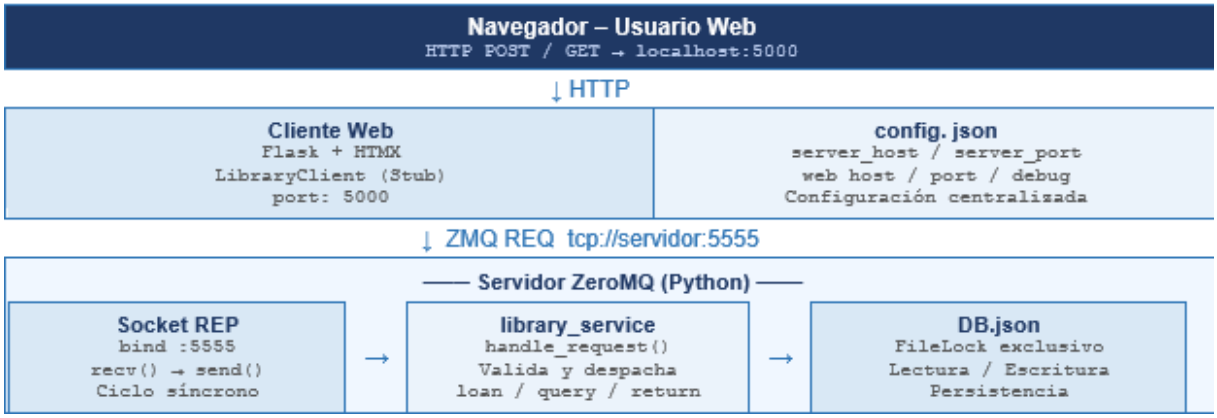

Figura 4.1: Arquitectura del sistema

4.2. Componentes

4.2.1. Modelo de Datos

La base de datos se definió en archivo de formato JSON(db.json). Cada libro en la base de datos tiene siete campos. Los primeros tres (ISBN, título y autores) no cambian. Los últimos cuatro (estado, prestatario y fechas) cambian cada vez que se hace un

préstamo o una devolución, como se muestra en la tabla. Un libro puede tener diferentes estados: **no prestado, prestatario=null, fecha prestamo=null, fecha devolucion=null o prestado, prestatario='Sofia', fecha prestamo='2026-02-24', fecha devolucion='2026-03-03'**.

Campo	**Tipo**	**£Vacío?**	**Descripción**
ISBN	Texto	No	Código único del libro (formato ISBN-13, ej. 9780307474278).
titulo	Texto	No	Nombre completo del libro.
autores	Lista de textos	No	Nombres de los autores del libro.
estado	Texto	No	prestado si el libro está en uso; no prestado si está disponible.
prestatario	Texto	Sí (si no está prestado)	Persona que tiene el libro actualmente.
fecha_prestamo	Fecha (AAAA-MM-DD)	Sí (si no está prestado)	Día en que se realizó el préstamo.
fecha_devolucion	Fecha (AAAA-MM-DD)	Sí (si no está prestado)	Fecha límite para la devolución.

Cuadro 4.1: Estructura del modelo de datos

4.2.2. Servidor

El servidor es el componente que ejecuta la lógica principal y toma las decisiones importantes. Cuando un usuario consulta, presta o devuelve un libro desde la interfaz web, la solicitud termina siendo procesada aquí. El servidor valida si el libro existe, si está disponible o ya está prestado, si se puede registrar una devolución, y si se deben actualizar fechas y estados. También es el único que modifica la base de datos. Esto garantiza que todas las operaciones pasen por un punto central de control y evita inconsistencias. El socket ZMQ REP permanece en escucha indefinida sobre el puerto 5555; cada mensaje recibido es procesado completamente antes de aceptar el siguiente, siguiendo el ciclo estricto del patrón REQ-REP: recv procesa send.

4.2.3. Cliente

El cliente es el intermediario entre la aplicación web y el servidor. No ejecuta la lógica del negocio; simplemente se encarga de enviar solicitudes al servidor y recibir respuestas. La web habla HTTP, el servidor habla ZMQ sobre TCP, y el cliente (LibraryClient) traduce entre ambos. La clase LibraryClient mantiene un socket REQ persistente durante la vida de la aplicación Flask. Cada llamada a un método construye el diccionario de solicitud, lo serializa a JSON, lo envía y espera la respuesta. Si el servidor no responde

dentro del tiempo de espera configurado, la excepción zmq.Again es capturada y se retorna un error descriptivo al usuario sin bloquear indefinidamente la aplicación.

Capítulo 5

Implementación

5.1. Tecnologías utilizadas

En la tabla se muestran las tecnologías empleadas para el desarrollo del sistema, así como su propósito dentro de la arquitectura general. La selección de herramientas se realizó considerando simplicidad, compatibilidad entre componentes y facilidad de implementación en un entorno distribuido.

Tecnología	**Versión**	**Para qué se usó**
Python	3.10+	Lenguaje principal del proyecto (servidor y cliente).
ZeroMQ (pyzmq)	$\geq$ 26.0.0	Mensajería entre el cliente web y el servidor de biblioteca.
Flask	$\geq$ 3.1.0	Servidor web que muestra la interfaz al usuario en el navegador.
HTMX	Última estable	Permite actualizar partes de la página sin recargar completamente.
JSON	Estándar	Formato de los mensajes entre cliente y servidor, y almacenamiento en la base de datos.

Cuadro 5.1: Tecnologías

5.2. Estructura

El proyecto se encuentra organizado en dos partes principales: el módulo del servidor y el módulo del cliente. Cada archivo cumple una función específica dentro de la arquitectura distribuida.

5.2.1. server/main.py

lee los parámetros definidos en el archivo de configuración y pone en funcionamiento el sistema de mensajería. Aquí se establece el puerto en el que el servidor escuchará solicitudes. En este archivo se pueden destacar las siguientes funciones.

- **load_config()** Carga el archivo de configuración config.json ubicado en el directorio raíz del proyecto y retorna su contenido como un diccionario de Python. Permite centralizar los parámetros de configuración del servidor (host, puerto). Usa la constante CONFIG_PATH, en la que se carga la información del .json. Retorna un diccionario con la configuración completa del sistema.

- **main()** Función principal del servidor, se encarga de leer la información del sistema, extraer los parámetros del servidor, construir la dirección de enlace ZeroMQ, mostrar información de arranque en consola e inicar el servicio principal llamando a la función **run_service()**.

5.2.2. server/library_service.py

contiene la lógica principal del sistema. En este se implementa el patrón REQ-REP en ZeroMQ y se definen las operaciones que el servidor puede ejecutar, como consultar, prestar o devolver libros. Funciones:

- **handle_request()** Se encarga de procesar solicitudes especificas de cliente relacionadas con operaciones sobre recursos del sistema, siendo estas, prestamo por ISBN, Prestamo por título, consulta por ISBN y devolución por ISBN. Recibe un mensaje o request, extrae los parámewtros de este, verifica que sea una solicitud admitida por el servidor y llama a las funciones que manejan estas consultas para enviar la correspondiente respuesta, representada en el retorno de dichas funciones.

- **loan_by_title/isbn** Recibe el título o el ISBN del libro según el caso, procesa la solicitud y le indica al usuario si el libro fue encontrado, si se pudo realizar el préstamo o si no.

- **_query_by_isbn()** Recibe el ISBN de un libro y consulta su disponibilidad, devolviendole la respuesta al usuario.

- **_return_by_isbn()** Recibe el ISBN de un libro y realiza la devolución en la base de datos, enviándole la respueta al usuario.

- **run_service()** Función principal que inicializa y ejecuta el servidor zeroMQ utilizando el patrón: REQ - REP implementando concurrencia real mediante múltiples procesos. Recibe como parámetros la dirección TCP donde el servidor escucha peticiones. Flujo:

  1. Crea el contexto principal ZMQ.
  2. Crea el manejo de frontent y backend.
  3. Calcula puerto backend interno.
  4. Llama a la función que enruta mensajes automáticamente.
  5. Finaliza todo terminando procesos, cerrando sockets y finalizando el contexto.

5.2.3. server/db.py

se encarga de la gestión de la base de datos. Su responsabilidad es leer y escribir el archivo DB.json, que almacena la información de los libros. Funciones:

- **__read__db():** Lee el archivo DB.json y retorna la lista de libros almacenados. Retorna una lista de libros.
- **__write__db():** Sobreescribe el archivo DB.json con la lista actualizada del estado de los libros Recibe la lista completa de los libros.
- **get__all__books/__book__by__isbn/__book__by__title:** Funciones de consulta que permiten acceder a información de la base de datos sin modificarla, usan _file_lock y llaman a _read_db() Reciben el ISBN o el título del libro cuando aplica. Retorna una lista con los libros encontrados.
- **loan__book/__by__title:** Funciones para el préstamo de libros, verifican su existencia, disponibilidad y posteriormente actualizan estado, guardan fechas y persisten los cambios realizados a la base de datos. Reciben el isbn o el título y el nombre de quién pide el libro. Retorna un bool para indicar éxito o fallo, un mensaje descriptivo y los datos del libro en caso de que haya sido éxitoso.
- **return__book():** Procesa la devolución de un libro, para esto verifica existencia, valida que esté prestado, restaura el estado a disponible, limpia cambios de pestamo y guarda todo en la base de datos. Recibe el ISBN del libro. Retorna un tupla indicando éxito o fallo y un mensaje descriptivo.

5.2.4. client/app.py

corresponde al servidor web desarrollado con Flask. Este módulo recibe las acciones del usuario desde el navegador, procesa los formularios y envía las solicitudes al servidor mediante el cliente de mensajería. También es responsable de renderizar las plantillas HTML y devolver la respuesta al navegador. Funciones:

- **__load__web__config():** Carga la configuración del servidor web desde el archivo config.json, únicamente de la sección "web", para parametrizar host del servidor, puerto y modo debug, usa la constante CONFIG_PATH. Retorna un diccionario con la configuración del servidor o uno vacío si ocurre algún error.
- **index():** Controlador de la página principal, renderiza la vista base index.html que conteine las cuatro operaciones disponibles en el sistema. Retorna el HTML renderizado.
- **Operaciones de biblioteca:** incluye a las funciones loan_isbn, loan_title, query_isbn y return_isbn, usadas como controladores HTTP. Se usan obteniendo información del formulario, invocando el cliente ZeroMQ y renderizando una plantilla parcial con el resultado. Retornan el HTML renderizado

5.2.5. client/zmq__client.py

gestiona la comunicación con el servidor. Aquí se empaquetan las solicitudes en formato JSON y se envían a través del sistema de mensajería. Funciones:

- **__default__server__address():** Obtiene la dirección del servidor ZeroMQ leyendo los parámetros de configuración desde el archivo config.json, accediendo a la sección client y extrayendo las secciones server_host y server_port. Retorna la dirección del servidor en formado ZeroMQ.

Este script contiene la clase LibraryClient, que encapsula toda la lógica de comunicación entre el cliente y el servidor usando ZeroMQ bajo el patrón request-reply. Métodos:

- **___init___():** Constructor de la clase, inicializa la conexión con el servidor ZeroMQ. Flujo:

  1. Si no se proporciona una dirección de servidor llama a _default_server_address() para obtenerla.
  2. Crea un contexto ZeroMQ.
  3. Crea un socket tipo REQ.
  4. Conecta uel socket a la dirección del servidor.
  5. Configura los tiempos de espera.

  Recibe como entrada la dirección del servidor, de manera opcional.

- **close():** Cierra correctamente la conexión con ZeroMQ, cerrando el socket y terminando el contexto ZeroMQ.

- **_send_request():** Método interno que envía una petición al servidor y espera la respuesta. Flujo:

  1. Convierte el diccionario request a json y lo codifica en bytes.
  2. Envía los datos al servidor mediante el socket.
  3. Espera la respuesta.
  4. Decodifica la respuesta de bytes a string.
  5. Convierte el json recibido a diccionario.

  Maneja errores de la siguiente forma: zmq.Again, cuando el servidor tarda mucho en responder, zmq.ZMQError cuando hay un error de conexión, Exception cuando ocurre un error inesperado. Recibe un diccionario con la acción y parámetros necesarios. Retorna la respuesta del servidor.

- **Operaciones de biblioteca;** Construyen la estructura del mensaje y delegan el envío a _send_request(). Incluye a los métodos loan_by_isbn() y loan_by_title() que solicitan el préstamo de un libro usando su ISBN o su título, query_by_isbn() que consulta la información de un libro por su ISBN y return_by_isbn que solicita la devolución de un libro.

5.2.6. client/templates/

contiene las páginas HTML que el usuario visualiza en el navegador. Estas plantillas definen la estructura de la interfaz gráfica y permiten la actualización dinámica de contenido.

5.2.7. config.json

centraliza los parámetros del sistema. En él se definen los puertos de comunicación y las direcciones IP. Esto permite modificar el entorno sin cambiar el código fuente.

5.2.8. Flujo de una petición

Cada vez que un cliente realiza cualquier petición al servidor se sigue el siguiente proceso:

1. Usuario envía formulario utilizando Flask.
2. Flask llama a las funciones de client para comunicarse con el servidor.
3. El cliente construye un JSON que contiene la solicitud, y lo codifica a binario.
4. El cliente envía el mensaje por el socket REQ utilizando ZeroMQ.
5. El servidor REP recibe la solicitud.
6. Se ejecuta la lógica del servidor para procesar la solicitud correctamente.
7. Se usa FileLock para acceder de forma segura a la base de datos JSON.
8. El servidor genera su respuesta y la codifica.
9. El cliente recibe la respueta del servidor.
10. Flask le muestra la respuesta al cliente.

Capítulo 6

Diseño de Experimentos

6.1. Entorno de Trabajo

Todas las pruebas se realizaron en computadores del laboratorio de la Pontificia Universidad Javeriana conectados bajo la misma red institucional. A continuación las especificaciones de los equipos, obtenidas con el comando top directamente del sistema operativo

Característica	**Servidor y cliente 1**	**Cliente 2**
Cantidad de núcleos	4 cores	24 cores
Memoria RAM	8GB	32GB
Sistema Operativo	Ubuntu	Windows 11.

Cuadro 6.1: Equipos utilizados

6.2. Base de datos

Para poder verificar la eficacia de los experimentos, el equipo diseñó una base de datos usando json. Dicha base de datos debe evidenciar los cambios realizados por los clientes luego de efectuar sus solicitudes al servidor, viendose reflejados automáticamente luego de hacer un préstamo o una devolución. En este espacio se cuenta con la siguiente información:

- ISBN único de los libros.
- Título del libro.
- Autor o autores del libro.
- Estado del libro: prestado/no prestado.
- prestatario.
- fecha_prestamo.
- fecha_devolución.

Cuando se realiza un préstamo, se debe cambiar el estado del libro a "prestado", guardar el nombre del prestatario y las fechas de prestamo y devolución. Cuando se devuelve el libro, todos los datos anteriores deben ser reemplazados por "nullz el estado devuelto a "no prestado". Que esta información sea actualizada correctamente determina si el programa está funcionando correctamente o no.

6.3. Pruebas preliminares

Antes de realizar los experimentos usando varios equipos diferentes, se simula el modelo cliente-servidor con un solo equipo, desde dos terminales diferentes, conectados mediante la IP única de la máquina, de tal forma se logra validar la eficacia del programa antes de pasar a las pruebas de comunicación remota, en las que se centra el trabajo realizado.

6.4. Pruebas de Comunicación Remota

Las pruebas de comunicación remota tienen como objetivo validar el correcto funcionamiento del intercambio de información entre los clientes y el servidor, en un entorno distribuido usando ZeroMQ bajo el patrón REQ-REP, con estas pruebas se busca evaluar correcta conexión entre equipos, integridad de los mensajes enviados y comportamiento ante múltiples clientes.

6.4.1. Entorno de pruebas

- En primer lugar se debe ejecutar el servidor, utilizando el comando **python -m server.main**, desde este equipo se mantiene todo el servicio de biblioteca que envia respuestas a las solicitudes de préstamo o de consulta.
- Se ejecuta el programa para los clientes en uno o mas equipos utilizando el comando **python -m client.app**, esto ejecuta el cliente ZMQ y la aplicación web en Flask.
- La comunicación de estos dos componentes se realiza mediante protocolo TCPI/IP con el puerto configurado previamente.

6.4.2. Prueba de conectividad inicial

Para verificar que el cliente puede establecer conexión remota con el servidor, se inicia el servidor, luego se incia el cliente y finalmente se ejecuta una consulta simple, esperando que el cliente obtenga una respuesta válida.

6.4.3. Prueba de envío y recepción

Desde un solo cliente, se prueba que el servidor reciba correctamente solicitudes y responda como fue planteado en el código, para esto se deben realizar todas las acciones que admite el servidor:

1. Consulta por ISBN
2. Consulta por Título

3. Préstamo por ISBN
4. Préstamo por título
5. Préstamo a un libro no disponible
6. Devolución

En base a lo anterior, se analiza si las respuestas dadas por el servidor son consistentes con lo plasmado en la base de datos, y si sus actualizaciones a la misma son efectivas.

6.4.4. Pruebas de concurrencia

Para evaluar el comportamiento del servidor bajo múltiples conexiones simultáneas, se deben repetir las pruebas anteriores, pero desde dos equipos diferentes, intercalando en quien realiza los préstamos y realizando préstamos en paralelo. De tal forma se verifica consistencia en la base de datos del servidor y manejo de concurrencia mediante el oportuno procesamiento de solicitudes y respuestas, forzando al servidor a solo procesar un formulario a la vez para evitar errores en la sobreescritura de información y en la información compartida con el cliente. Se establecieron protocolos que permitían asegurar integridad en los procesos y en la base de datos, de tal forma que no se presenten errores al efectuar una acción

Capítulo 7

Análisis de Resultados

7.1. Conexión en una misma máquina

Para las pruebas de conexión en una misma máquina se obtuvieron los siguientes resultados desde la página web correspondiente al cliente:

Figura 7.1: Consulta desde un mismo equipo

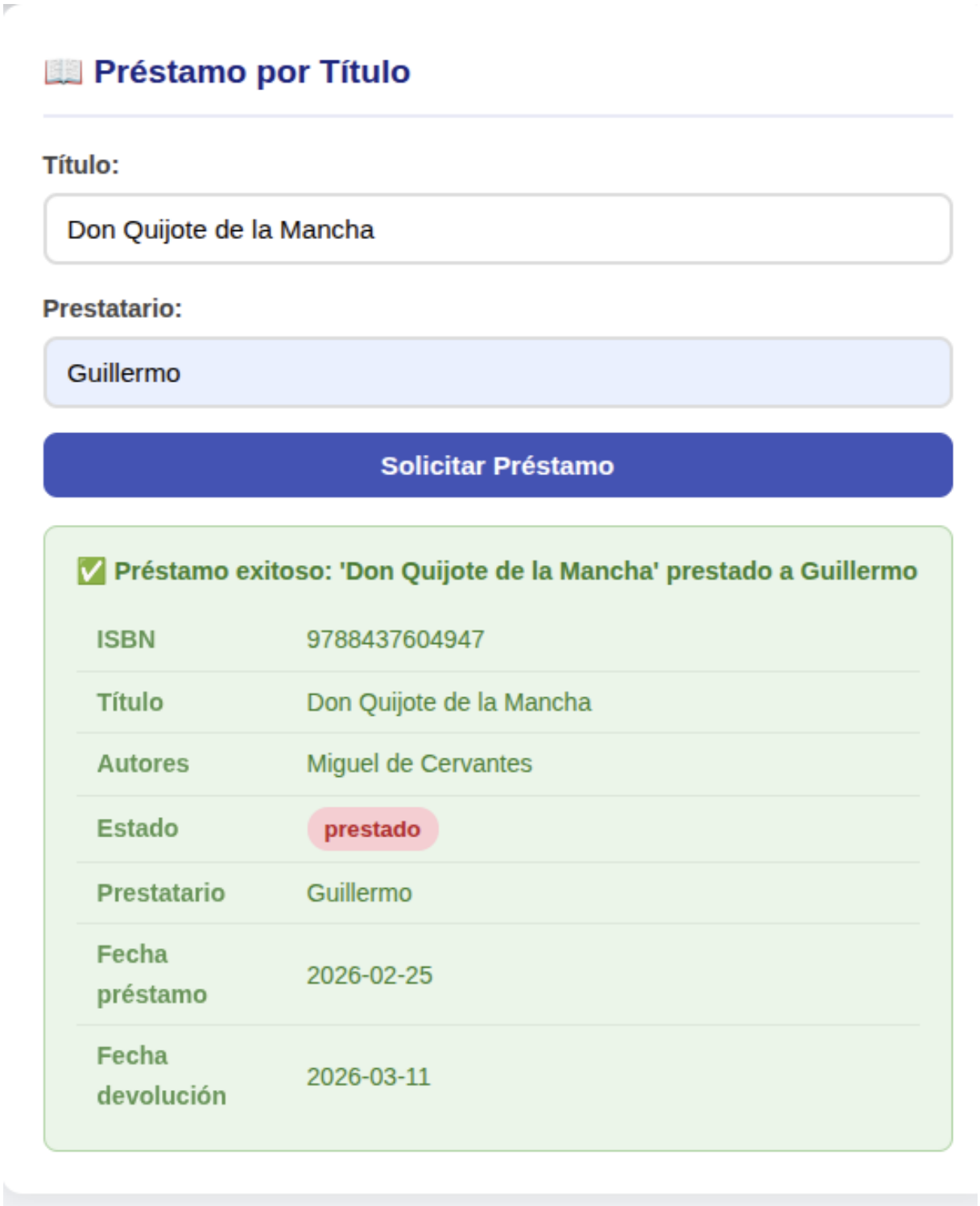

Figura 7.2: Prestamo por título desde un mismo equipo

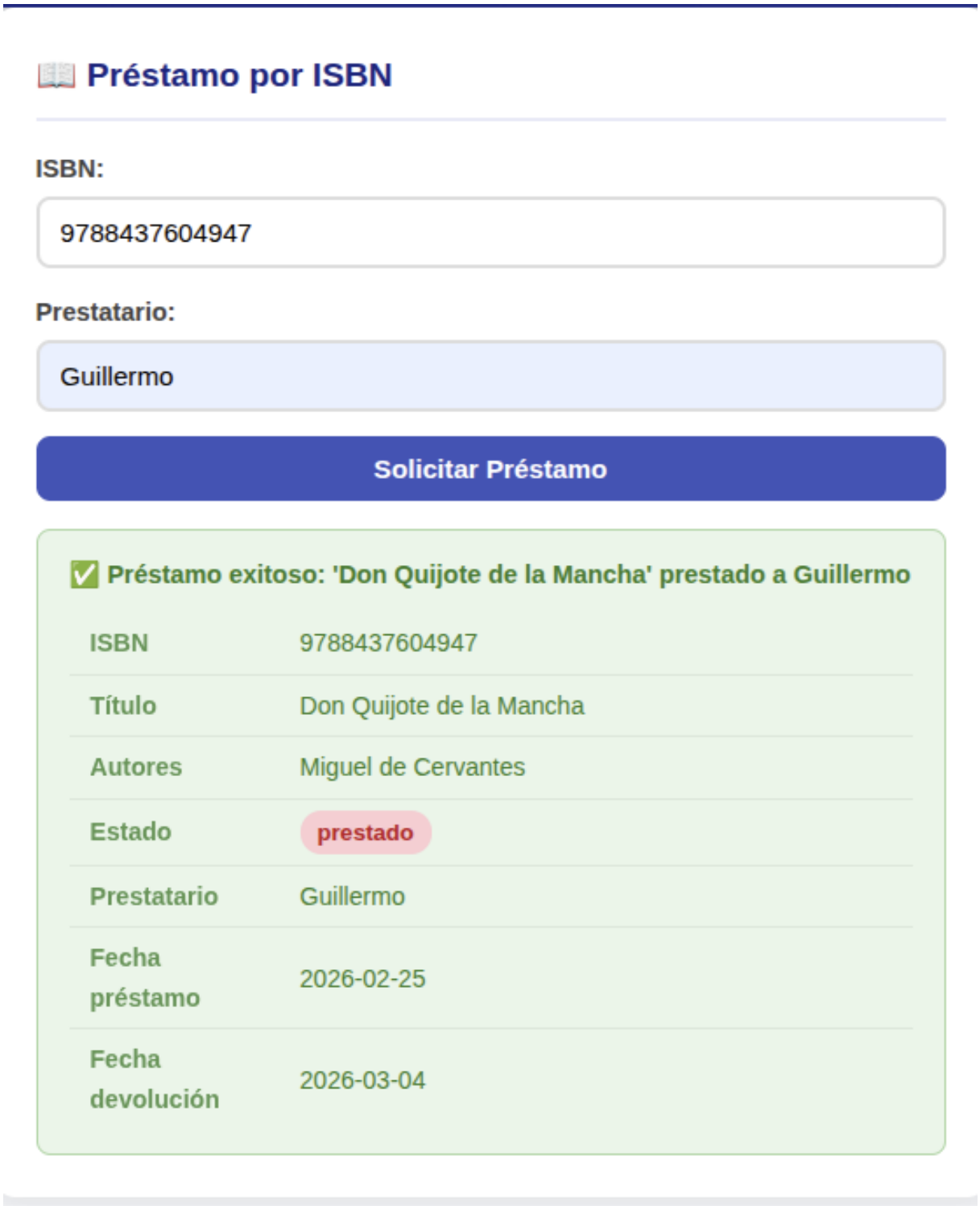

Figura 7.3: Prestamo por ISBN desde un mismo equipo

Figura 7.4: Devolución desde un mismo equipo

En la siguiente imágen se puede evidenciar como la base de datos efectivamente se actualiza después de cualquier solicitud:

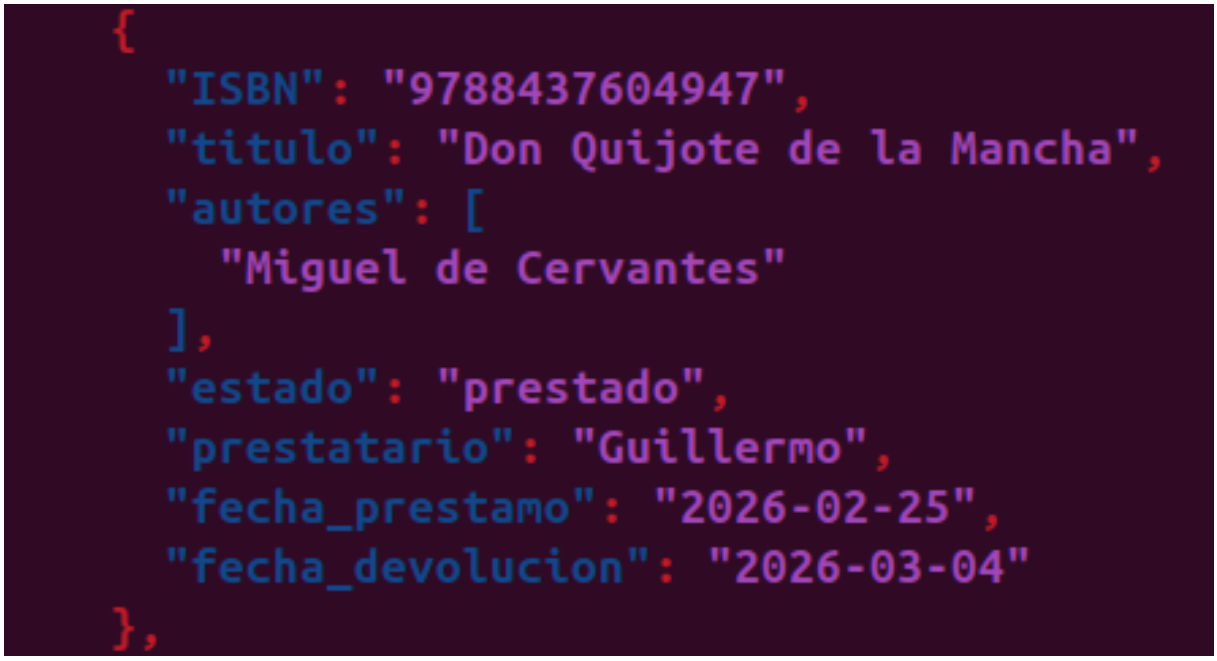

Figura 7.5: Actualización de la base de datos desde un mismo equipo

Finalmente, se evidencia como el servidor procesa correctamente las solicitudes realizadas por los clientes:

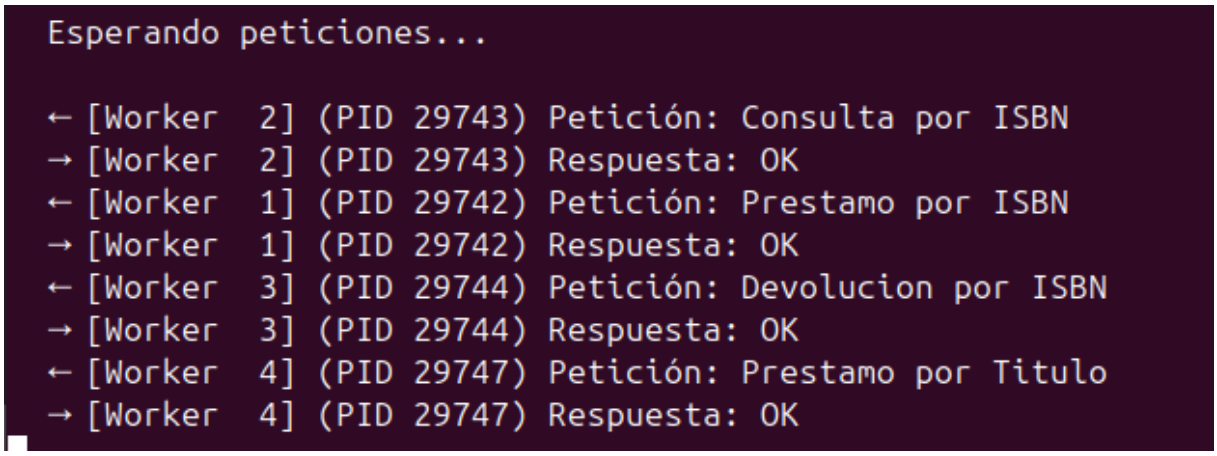

Figura 7.6: Procesamiento de solicitudes desde el servidor

7.2. Conexión en varias máquinas

Una vez verificado que la conexión era efectiva en una sola máquina, se pasa a realizar las pruebas usando equipos diferentes para cumplir el papel de cliente y servidor.

En primer lugar, se evidencia como cualquier préstamo se realiza correctamente, desde una máquina cliente, visible en la siguiente figura:

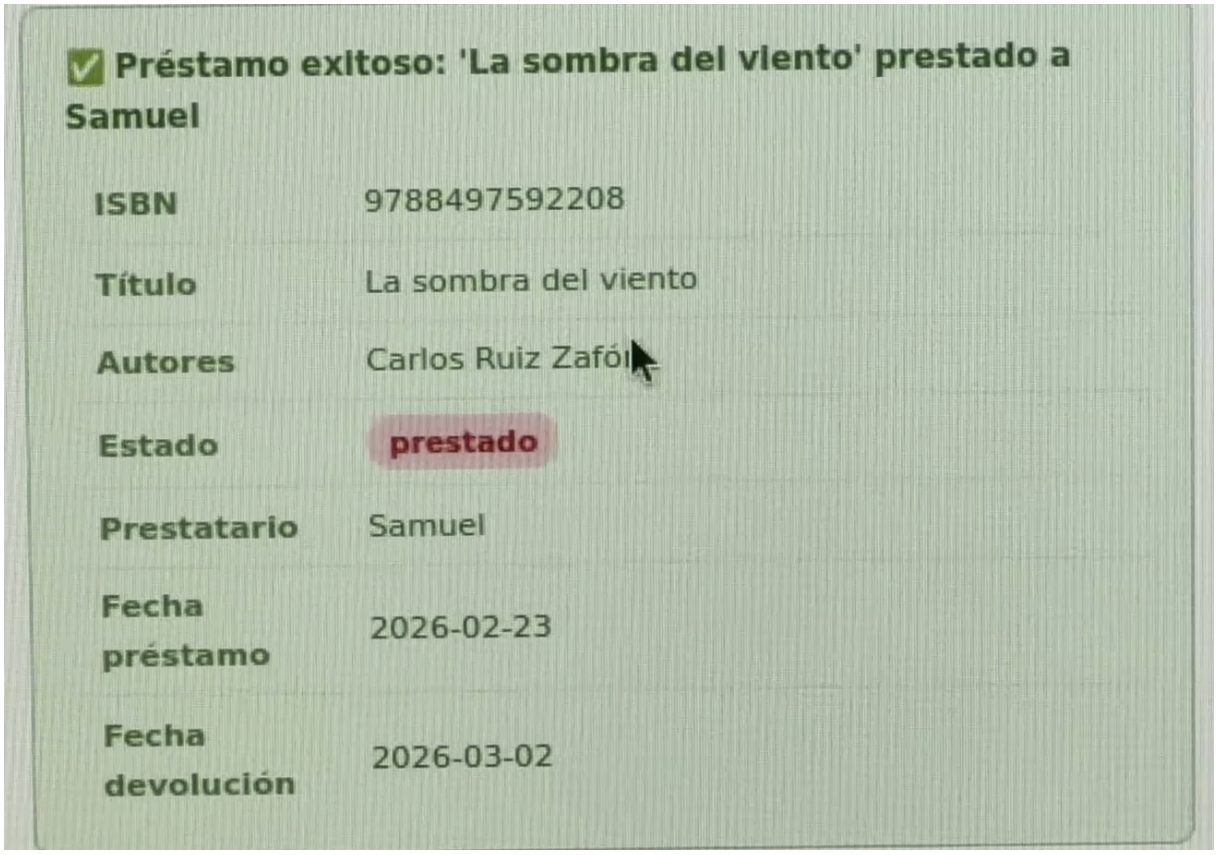

Figura 7.7: Prestamo correcto desde el cliente

Cuando se intenta realizar el préstamo al mismo libro, pero desde un segundo cliente, se logra ver como el préstamo es rechazado, pues el primer cliente ya posee el libro.

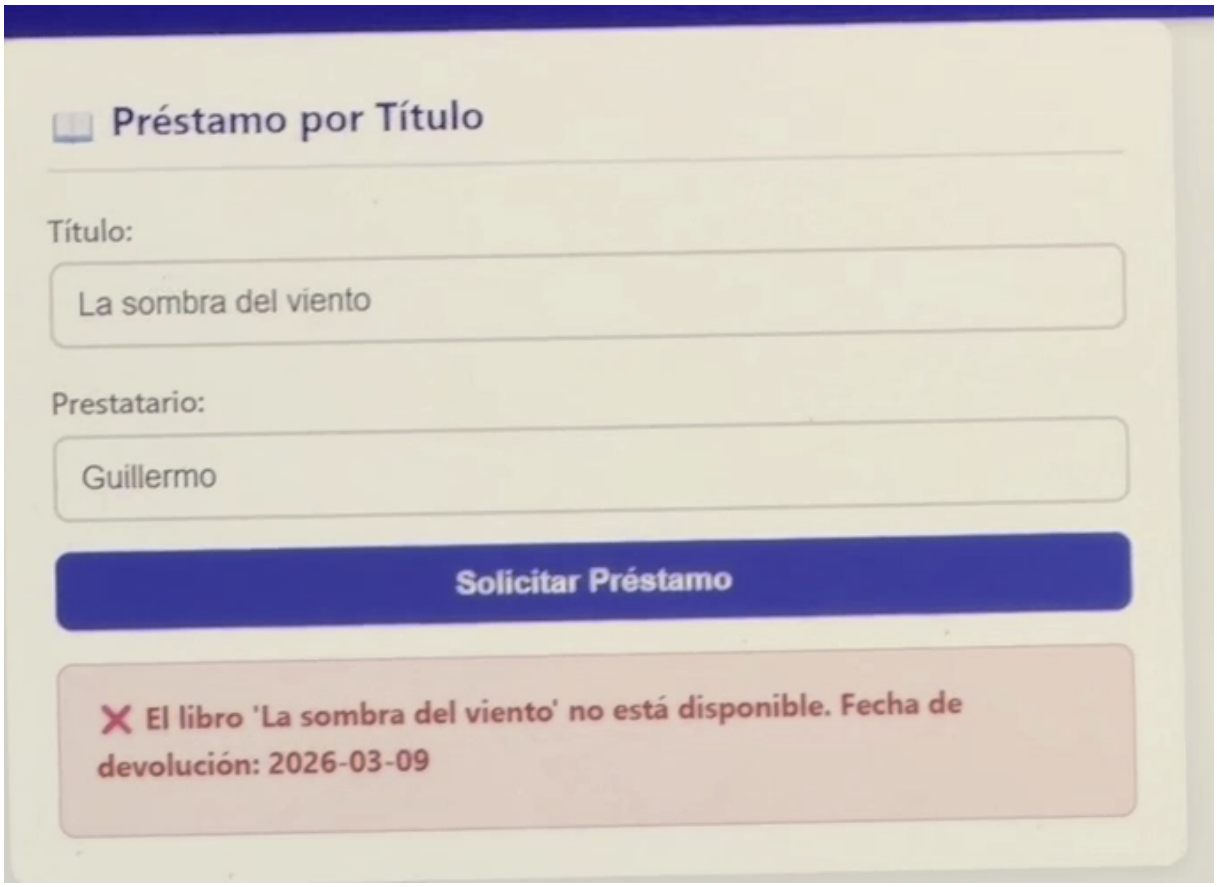

Figura 7.8: Prestamo rechazado

Si desde el segundo cliente, se solicita un libro diferente que si esté disponible, el servidor acepta la solicitud.

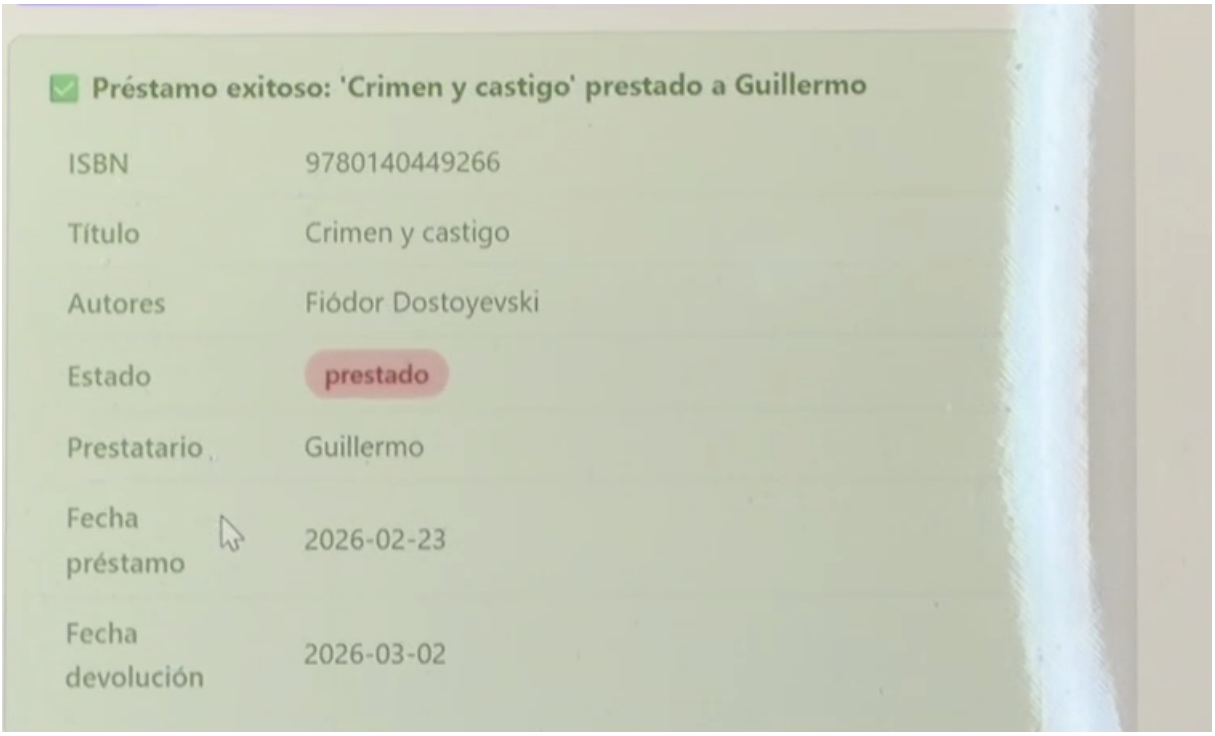

Figura 7.9: Prestamo aceptado

Finalmente, se logra evidenciar como el servidor es capaz de procesar todos las solicitudes simultáneas que son enviadas desde los clientes.

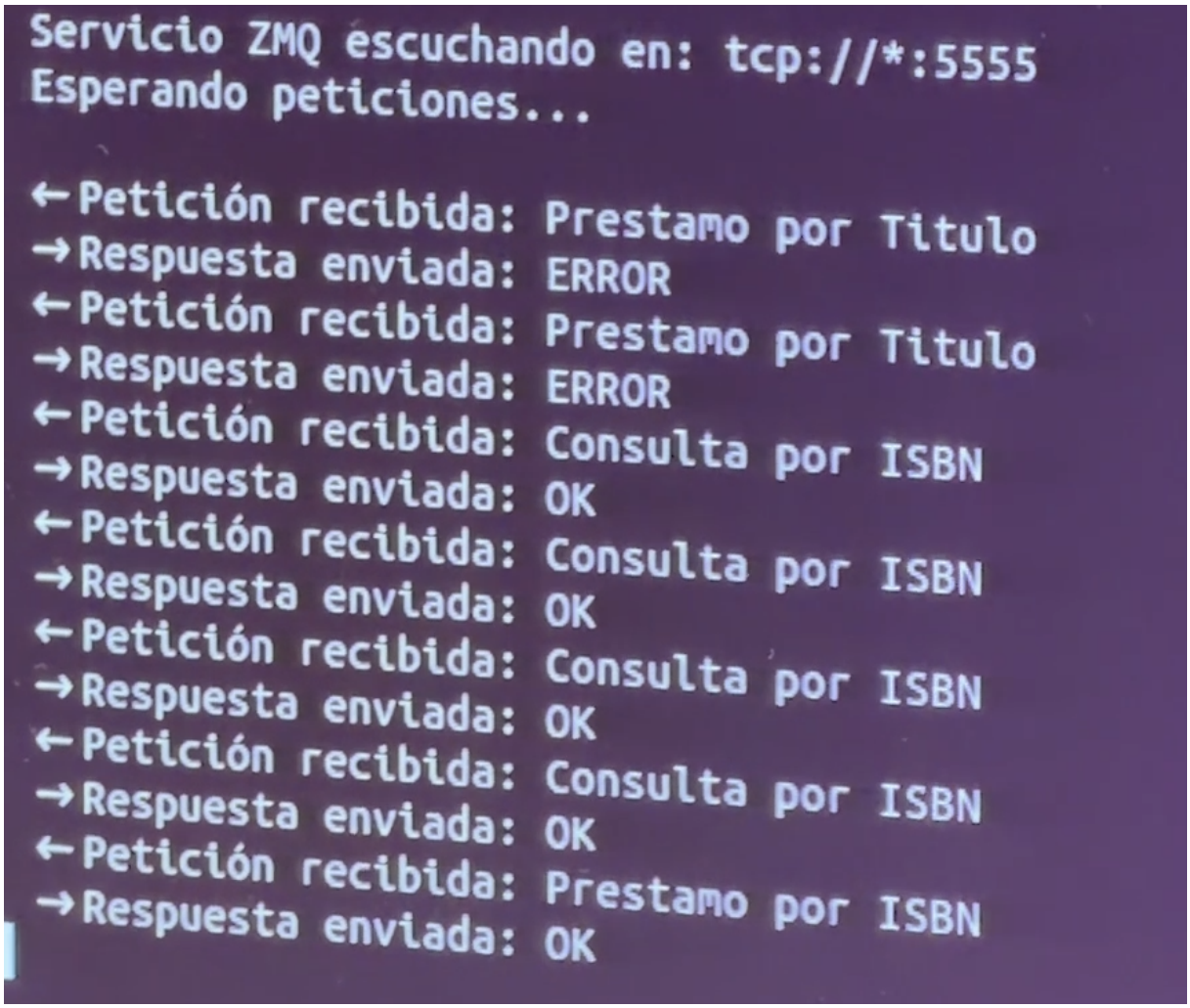

Figura 7.10: Procesamiento de solicitudes desde el servidor

7.3. Latencia y estabilidad

Se logra ver como el servidor, siempre que esté conectado a una red, se mantiene activo indefinidamente, al igual que con los clientes, capaces de seguir ejecutando el proceso hasta que un factor externo, como el usuario o fallas inevitables en la conexión intervengan, cumpliendose el propósito del trabajo. Puesto que las tareas realizadas por el sistema son simples, no representan una gran cantidad de datos, y son realizadas en un espacio controlado los tiempos de respuesta del servidor resultan casi imperceptibles, tanto en pruebas con un solo equipo como en pruebas con varios, logrando evidenciar la eficacia del sistema producido para la actividad, notándose además como absolutamente todas las solicitudes recibieron una respuesta equivalente a lo solicitado, sin temas de desincronización o mensajes duplicados. En cuanto a la conexión, se puede concluir que se hizo una excelente coordinación en todas las capas del sistema, aplicando correctamente las herramientas establecidas: json para el manejo de bases de datos, Python como lenguaje de backend. flask como framework web y zeroMQ para la comunicación distibuida de los procesos

7.4. Integridad de los datos

Durante los procesos de guardado, envío y lectura de datos, se logró determinar que todos los datos llegan desde el cliente al servidor sin sufrir ningún tipo de alteración, esto ayuda a mantener el formato JSON bien estructurado y sin ninguna pérdida de campos en la transmisión, esto demuestra nuevamente la efectividad de zeroMQ como una herramiento de comunicación entre procesos. Además del correcto envio de datos usando zeroMQ, el diseño y creación del código permite que el sistema no presente fallos ante los desafios que pudieran presentársele, esto gracias al correcto manejo de excepciones en caso de ingresar datos erroneos, inexistentes o peligrosos, además de el correcto manejo en la concurrencia de datos, al manejar ordenada y oportunamente todas las solicitudes del cliente, el servidor no deja solicitudes sin contestar, no da respuestas con datos erroneos y lo más importante, no trabaja mas de una solicitud a la vez, permitiendo contestar correctamente todas las peticiones que se le van enviando. Se consiguió que el sistema de respuestas claras antre cualquier problemática y reaccione sin poner en riesgo ningún componente lógico o físico utilizado en este.

7.5. Conclusiones técnicas del análisis

7.5.1. Sobre el patrón REQ-REP

La implementación de un patrón REQ-REP permitió solucionar correctamente el problema planteado para la actividad, pues, por sus carácteristicas asegura el correcto funcionamiento modelo cliente-servidor con varios equipos realizando solicitudes.

- Garantiza la síncronia estricta.
- Obliga a mantener el ciclo petición-respuesta.
- Crea consistencia para la lógica del servidor.
- Mejora la calidad estructural de las solicitudes y respuestas

7.5.2. Sobre concurrencia

Aunque el servidor procesa solicitudes una a una, se enfrentó el hecho de que múltiples pueden estar realizando peticiones concurrentemente, para evitar cualquier tipo de falla en estas solicitudes se optó por implementar una arquitectura robusta ante fallos y eficaz ante sus tareas, permitiendo que cada solicitud sea respondida como corresponde, sobre la manera en que se abordó esta característica de la solicitud se puede concluit lo siguiente.

- Se demostró que múltiples clientes pueden interactuar sin errores.
- No se presentaron condiciones de carrera.
- No hubo corrupción de la base de datos JSON.
- Se garantizó exclusión mútua entre todas las solicitudes gracias a como son procesadas.

7.5.3. Sobre estabilidad

Se logró mantener una comunicación estable y fiel ante las solicitudes realizadas por los clientes y las modificaciones en la base de datos derivadas de esto.

- No se evidenció pérdida de mensajes.
- No se observaron bloqueos indefinidos.
- No se generaron inconsistencias entre clientes.
- El sistema siempre mantuvo coherencia en todas sus acciones.

7.6. Resumen del análisis

- Se logró efectuar una conexión fiel, eficiente, estable y segura mediante el modelo cliente-servidor.
- Se logró cuidar la integridad de los datos enviados mediante zeroMQ y manipulados desde el servidor en la base de datos.
- Se logró asegurar un comportamiento robusto y coherente del sistema ante cualquier posible fallo.
- Se logró asegurar el correcto funcionamiento del servidor mediante la aplicación de procesos de concurrencia.
- Se logró verificar experimentalmente la efectividad de tecnologías para la distribución de procesos.

Capítulo 8

Discusión

8.1. Dificultades Encontradas

El desarrollo del sistema presentó varios desafíos técnicos que obligaron al equipo a revisar decisiones de diseño durante la implementación. El primero estuvo relacionado con la sincronización de la base de datos: en versiones tempranas, dos solicitudes casi simultáneas podían leer DB.json antes de que la primera escribiera su cambio, generando sobrescrituras indeseadas. Esto se resolvió al introducir FileLock, aunque encontrar la librería correcta compatible con Unix (fcntl) y Windows (msvcrt) tomó más tiempo del esperado. Otro punto de dificultad fue la configuración de red en el laboratorio. Las máquinas tienen restricciones de puertos que inicialmente impedían la conexión ZMQ entre equipos. Además, el ciclo estricto del patrón REQ-REP exige que el servidor siempre envíe una respuesta antes de poder recibir otra solicitud; cuando el servidor lanzaba una excepción sin responder, el socket REQ del cliente quedaba bloqueado indefinidamente. Esto obligó a blindar completamente el servidor con manejo de excepciones para garantizar que siempre se envía una respuesta, incluso ante errores. Finalmente, integrar HTMX con Flask de forma que las actualizaciones parciales fueran consistentes requirió definir con precisión qué fragmento HTML debía retornar cada ruta, especialmente cuando la respuesta podía ser un error o un éxito con estructuras de datos diferentes.

8.2. ZeroMQ vs gRPC

La elección de ZeroMQ sobre gRPC fue una de las decisiones más debatidas. gRPC ofrece ventajas en ergonomía de desarrollo: define el contrato en un archivo .proto, genera automáticamente los stubs del cliente y el servidor, y usa HTTP/2 para serialización eficiente y tipada. En un proyecto de mayor escala o con múltiples lenguajes de programación, gRPC sería probablemente la opción preferible. Sin embargo, ZeroMQ tiene características que se ajustan mejor a los objetivos de este taller. Al ser una biblioteca de comunicación de alto rendimiento que opera directamente sobre sockets, no depende de que el servidor conteste de inmediato para que el cliente pueda procesar internamente: ZeroMQ maneja el encolado de forma transparente a nivel de biblioteca. Más importante aún, ZeroMQ no impone ningún protocolo de aplicación, lo que obliga al desarrollador a entender y diseñar explícitamente la arquitectura de mensajería. Esto lo convierte en una herramienta más didáctica para un taller de sistemas distribuidos, donde el objetivo es comprender los mecanismos subyacentes y no solo usarlos de forma transparente.

Adicionalmente, la ausencia de un protocolo impuesto significa menos componentes que configurar en un entorno de laboratorio con restricciones de red.

8.3. Interfaz web con Flask y HTMX

La especificación del taller no requería una interfaz gráfica; bastaba con un cliente de línea de comandos. Sin embargo, el equipo decidió implementar la interfaz web porque facilita la prueba y validación de cada operación de forma visual y estetica, y hace que las demostraciones en el laboratorio sean más claras. Una CLI habría requerido recordar la sintaxis de cada comando y leer la salida en texto plano. La interfaz web, en cambio, permite ver en tiempo real cómo cambia el estado del libro después de cada operación, con colores diferenciados para 'prestado' y 'no prestado' y mensajes de error legibles. HTMX se eligió sobre frameworks más pesados como React precisamente porque permite esta experiencia dinámica sin escribir JavaScript, manteniendo el proyecto completamente en Python en el backend.

8.4. Tolerancia a Fallos

Desde el diseño inicial se decidió que el servidor debía responder siempre, incluso ante errores. Esto es crítico en el patrón REQ-REP: si el socket REP no responde, el socket REQ del cliente queda bloqueado indefinidamente. Por esto, todo el cuerpo de **handle_request()** está envuelto en tryexcept, y cualquier excepción resulta en una respuesta JSON con success: false y un mensaje. Del lado del cliente, se configuraron timeouts en el socket REQ para capturar zmq.Again, retornando un error controlado en lugar de bloquear la aplicación Flask.

Capítulo 9

Conclusiones

El desarrollo del sistema de gestión de biblioteca permitió trasladar los fundamentos teóricos de los sistemas distribuidos a una implementación práctica y funcional. A través del uso de ZeroMQ como middleware de comunicación, se logró construir una arquitectura cliente-servidor en la que los componentes operan de manera desacoplada, pero coordinada, demostrando que la separación entre procesos y la distribución en red no impiden la coherencia ni la eficiencia del sistema cuando existe un protocolo bien definido.

La estandarización del intercambio de mensajes entre cliente y servidor fue un elemento clave para el correcto funcionamiento del sistema. Al definir un formato claro de comunicación, se garantizó que las operaciones de consulta, préstamo y devolución pudieran ejecutarse de manera consistente independientemente del equipo desde el cual se realizaran. Este diseño reafirma la importancia de establecer contratos de comunicación explícitos en sistemas distribuidos, donde la claridad del protocolo es tan relevante como la lógica interna del programa.

En cuanto a la atención de múltiples solicitudes, el servidor fue capaz de gestionar diferentes interacciones provenientes de varios clientes dentro de la red, manteniendo la estabilidad del sistema y la correcta ejecución de las operaciones. El uso de ZeroMQ facilitó un modelo de mensajería eficiente que permitió recibir y responder solicitudes de manera ordenada, evidenciando cómo la infraestructura de comunicación puede soportar escenarios de acceso concurrente sin comprometer la operatividad general.

Respecto al manejo de los recursos compartidos, el diseño centralizado del servidor permitió mantener el control del estado del catálogo, evitando inconsistencias en las operaciones realizadas por distintos usuarios. La gestión estructurada de las solicitudes aseguró que los datos se conservaran íntegros a lo largo de la ejecución del sistema, reforzando uno de los principios esenciales en entornos distribuidos: la preservación de la consistencia frente a múltiples accesos.

En conjunto, el taller demostró que un sistema distribuido no se limita a conectar múltiples máquinas, sino que requiere decisiones de diseño orientadas a la comunicación clara, el control del acceso y la integridad de la información. La experiencia permitió comprender de manera aplicada cómo los conceptos de cliente-servidor, concurrencia y control de recursos se materializan en una solución concreta, fortaleciendo la comprensión técnica y la capacidad de implementar arquitecturas distribuidas funcionales en contextos reales.

Bibliografía

[1] Hintjens, P. (2013). *ZeroMQ: Messaging for Many Applications. O'Reilly Media.*

[2] Grinberg, M. (2018). *Flask Web Development: Developing Web Applications with Python (2nd ed.). O'Reilly Media.*

[3] ZeroMQ Organization. (2024). *ZeroMQ Documentation.* Recuperado de https://zeromq.org/documentation/

[4] ØMQ - The Guide. (2025). *ZeroMQ Documentation.* Recuperado de https://htmx.org/ Recuperado de https://zguide.zeromq.org/