UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

DIVISIÓN DE INGENIERÍA ELÉCTRICA

INGENIERÍA EN COMPUTACIÓN

Sistemas Distribuidos

Profesor: Ing. José Abraham Bonilla Pastor

Proyecto final: “Sistema de gestión de salas de emergencias en un hospital”

Integrantes del equipo:

-Juárez Andrade Axel Yael

-Molina Véjar Aarón Gael

-Sánchez Gachuz Jennyfer Estefanya

-Zúñiga Mosco Rodrigo

Fecha: 02/12/2025

Índice

Contenido

Introducción

Este documento tiene como finalidad abarcar los antecedentes teóricos fundamentados en bibliografía importante dentro del ámbito de los sistemas distribuidos, las distintas etapas de desarrollo que llevaron al éxito de la implementación de un sistema distribuido que atiende a un entorno y modelo de negocio tan delicado como lo es un conjunto de salas de emergencias de un hospital.

Se espera cubrir todos los aspectos técnicos y de diseño de sistemas distribuidos que ayuden a entender la complejidad, funcionalidad y confiabilidad del proyecto, basándonos en objetivos, etapas y conceptos fundamentales de ingeniería de software y bases de datos englobados a los sistemas distribuidos.

Antecedentes

Redes de datos: Protocolo TPC

TCP (Protocolo de Control de Transmisión, del inglés Transmission Control Protocol) se diseñó específicamente para proporcionar un flujo de bytes confiable de extremo a extremo a través de una interred no confiable. Una interred difiere de una sola red debido a que sus diversas partes podrían tener diferentes topologías, anchos de

banda, retardos, tamaños de paquete y otros parámetros. TCP se diseñó para adaptarse de manera dinámica a las propiedades de la interred y sobreponerse a muchos tipos de fallas. [1]

El servicio TCP se obtiene al hacer que tanto el servidor como el receptor creen puntos terminales, llamados sockets, Cada socket tiene un número (dirección) que consiste en la dirección IP del host y un número de 16 bits que es local para ese host, llamado puerto. Un puerto es el nombre TCP para un TSAP. Para obtener el servicio TCP, hay que establecer de manera explícita una conexión entre un socket en una máquina y un socket en otra máquina. [1]

Bases de datos distribuidas

Los sistemas de bases de datos distribuidos están formados por sitios débilmente acoplados que no comparten ningún componente físico. Además, puede que los sistemas de bases de datos que se ejecutan en cada sitio sean sustancialmente independientes entre sí. La diferencia principal entre los sistemas de bases de datos centralizados y los distribuidos es que, en los primeros, los datos residen en una única ubicación, mientras que en los segundos los datos se reparten entre varios lugares. Esta distribución de los datos provoca muchas dificultades en el procesamiento de las transacciones y de las consultas.[2]

Almacenamiento distribuido de datos

Considérese una relación r que debe almacenarse en una base de datos. Existen dos enfoques de almacenamiento de esta relación en la base de datos distribuida:

- Replica: el sistema conserva varias replicas (copias) idénticas de la relación y guarda cada replica en un sitio diferente. La alternativa a las replicas es almacenar sólo una copia de la relación r
- Fragmentación: el sistema divide la relación entre varios fragmentos y guarda cada fragmento en un sitio diferente.[2]

En estos distintos enfoques haremos uso de una base de datos basada en réplicas, debido a la arquitectura diseñada.

Replicación

Hay dos razones principales para replicar datos: fiabilidad y rendimiento. Primero, los datos se replican para aumentar la fiabilidad de un sistema. Si un sistema de archivos ha sido replicado, puede ser posible continuar trabajando después de que una réplica falle simplemente cambiando a una de las otras réplicas. Además, al mantener múltiples copias, se vuelve posible ofrecer mejor protección contra datos corruptos. Por ejemplo, imaginemos que hay tres copias de un archivo y que cada operación de lectura y escritura se realiza en cada copia. Podemos protegernos contra una única operación de escritura que falle, considerando como correcto el valor devuelto por al menos dos copias.[4]

La otra razón para replicar datos es el rendimiento. La replicación para el rendimiento es importante cuando el sistema distribuido necesita escalar en número y área geográfica.[4] En general, la réplica mejora el rendimiento de las operaciones de lectura y aumenta la disponibilidad de los datos para las transacciones de lectura. Sin embargo, las transacciones de actualización suponen una mayor sobrecarga.[2]

Exclusión mutua

Los procesos distribuidos a menudo necesitan coordinar sus actividades. Si un conjunto de procesos comparte un recurso o una colección de recursos, a menudo se requiere exclusión mutua para evitar interferencias y garantizar la consistencia al acceder a los recursos. Este es el problema de la sección crítica, conocido en el ámbito de los sistemas operativos. Sin embargo, en un sistema distribuido, ni las variables compartidas ni las facilidades proporcionadas por un único núcleo local pueden utilizarse para resolverlo, en general. Necesitamos una solución para la exclusión mutua distribuida: una que se base únicamente en el paso de mensajes.[3]

Para nuestro sistema, la exclusión mutua va a tener un uso muy importante en la asignación de médicos y de camas disponibles, estos últimos serán nuestros recursos por lo que nuestros pacientes competirán por acceder a un recurso. Si los recursos se acaban, no se asignarán a clientes.

Tolerancia a fallos

Para entender el papel de la tolerancia a fallos en los sistemas distribuidos, primero necesitamos examinar más de cerca lo que realmente significa que un sistema distribuido tolere fallos. Ser tolerante a fallos está estrechamente relacionado con lo que se llaman sistemas confiables. La confiabilidad es un término que abarca una serie de requisitos útiles para los sistemas distribuidos, incluyendo los siguientes:[4]

- Disponibilidad
- Confiabilidad
- Seguridad
- Mantenibilidad

Elección

Un algoritmo para elegir un proceso único que desempeñe un rol particular se llama algoritmo de elección. Se necesita un algoritmo de elección para esta selección. Es esencial que todos los procesos estén de acuerdo con la elección. Posteriormente, si el proceso que desempeña el rol de servidor desea retirarse, se requiere otra elección para escoger un reemplazo. Decimos que un proceso llama a la elección si realiza una acción que inicia una ejecución particular del algoritmo de elección. [3]

Transparencia

No se debe exigir a los usuarios de los sistemas distribuidos de bases de datos que conozcan la ubicación física de los datos ni el modo en que se puede acceder a ellos en cada sitio local concreto. Esta característica, denominada transparencia de los datos, puede adoptar varias formas:[1]

- Transparencia de la fragmentación
- Transparencia de la replicación
- Transparencia de la ubicación

Los elementos de datos (como las relaciones, los fragmentos y las réplicas) deben tener nombres únicos. Esta propiedad es fácil de asegurar en las bases de datos centralizadas. En las bases de datos distribuidas, sin embargo, hay que tener cuidado para asegurarse de que dos sitios no utilicen el mismo nombre para elementos de datos diferentes.[1]

Transacciones distribuidas

El acceso a los diferentes elementos de datos en los sistemas distribuidos suele realizarse mediante transacciones, que deben preservar las propiedades ACID. Se deben considerar dos tipos de transacciones. Las transacciones locales son las que acceden a los datos y los actualizan en una única base de datos local; las transacciones globales son las que acceden a los datos y los actualizan en varias bases de datos locales.[2]

Desarrollo

Arquitectura distribuida

El desarrollo del proyecto se plantea con una arquitectura hibrida, buscando los siguientes objetivos:

- Estructuralmente es P2P: Según Tanenbaum, en los sistemas P2P, todos los nodos son funcionalmente iguales y ejecutan el mismo software [4]. En este caso las 4 salas de emergencia tienen el mismo código (main.py). Cualquiera puede iniciar una conversación con cualquiera, No hay un “servidor central” dedicado en un rack especial; todos son pares.
- Funcionalmente es Cliente-Servidor con un nodo coordinador, para resolver problemas difíciles como la exclusión mutua y el ordenamiento de los eventos primero necesitamos jerarquía.
    - Elección dinámica a un nodo para que actúe como Servidor (Maestro)

- Tres nodos funcionan temporalmente como Clientes (Esclavos) del maestro para las operaciones de escritura

Conclusión conceptual: El sistema es una red P2P estructurada donde los roles de cliente y servidor son dinámicos y rotativos, no fijos por hardware.

SQLite para la construcción de la base de datos

Para la implementación del almacenamiento fue seleccionado el motor SQLite. A diferencia de los sistemas cliente-servidor tradicionales, tales como PostgreSQL o MySQL, SQLite es un motor de base de datos sin servidor y autocontenido. [5]

La decisión fue estratégica debido a los siguientes puntos clave:

- Simulación de arquitectura Shared-Nothing: Al ser SQLite un archivo local, obliga a que cada nodo del sistema gestione su propio almacenamiento físico de manera aislada. Esto representa fielmente un entorno geográfico real con servidores dispersos.
- Eficiencia de recursos: Permite desplegar múltiples nodos en un entorno virtualizado limitado sin la sobrecarga de memoria que implicaría implementar demonios de bases de datos activos.
- Portabilidad: La base de datos reside en un único archivo (nodo_X.db), facilitando la replicación, el respaldo y el despliegue.

No existe un repositorio central de datos, en su lugar la información es replicada y fragmentada lógicamente a través de los nodos. Cada nodo mantiene su propio archivo de datos local en una ruta interna. Se logra una consistencia aplicando un modelo de replicación activa en todas las operaciones de escritura recibidas por el nodo maestro.

Diseño del esquema relacional

El esquema se diseñó para garantizar integridad referencial distribuida y control de recursos importantes para el sistema, las tablas principales son:

- Nodos: Almacena la topología de red necesaria para el enrutamiento dinámico y la elección del líder.
- Pacientes: Entidad fuerte con identificadores únicos globales para evitar colisiones o fallos de consistencia en la replicación
- Doctores: Incluye control de concurrencia con el campo: carga_actual y capacidad_max
- Camas: Recurso critico de exclusión mutua
- Visitas: Tabla transaccional que vincula Paciente-Doctor-Cama.

Nodo maestro

Arquitectura de Middleware

El nodo maestro se diseño e implementó como un Middleware de orquestación, situado lógicamente entre los clientes (interfaz de usuario) y la capa de almacenamiento distribuida. Su principal función es abstraer la complejidad de la red y garantizar la consistencia de datos.

La implementación se realizó utilizando Python y la librería socket sobre el protocolo TPC/IP, desarrollando una capa de aplicación personalizada basada en mensajería JSON. Este diseño permite un completo desacoplamiento entre los clientes que desconocen la topología de la red o la ubicación física de los datos, solo interactúan con la interfaz. Además de una centralización de la lógica actuando como único punto de entrada para todas las operaciones.

Algoritmo de exclusión mutua centralizada

Uno de los requerimientos críticos es evitar la sobreasignación de recursos finitos. Para resolver la concurrencia en un sistema distribuido, se implementó un mecanismo de exclusión mutua situada en el nodo maestro.

Se gestiona la sección critica en el que el maestro implementa un semáforo lógico, que serializa las peticiones de asignación de recursos, creando un flujo de transacción atómica:

1. Adquisición del Lock
2. Verificación de estado
3. Asignación y Commit
4. Liberación del Lock

Este enfoque garantiza que ante múltiples solicitudes simultaneas o condiciones de carrera, el sistema mantenga una consistencia estricta y nunca asigne el mismo recurso dos veces.

Estrategia de replicación y consistencia

El maestro actúa como emisor en un modelo de replicación primaria. Siempre valida y persiste la transacción en su propio almacenamiento antes de propagarla. Esto asegura que el líder siempre tenga el estado actual. Tras un Commit local exitoso, el servicio de replicación difunde la instrucción SQL exacta a todos los nodos esclavos detectados en la topología.

Se implementan timeouts para evitar que la caída de un nodo esclavo bloquee la operación del sistema principal, favoreciendo la disponibilidad sin sacrificar consistencia.

Lógica de negocio y enrutamiento

Desde el maestro se encapsulan reglas de negocio importantes para el dominio hospitalario, transformando todas las peticiones abstractas en operaciones de bases de datos concretas.

- Balanceo de carga de personal: Se implementa lógica para asignar pacientes a doctores basándose en su capacidad actual, optimizando así la distribución de trabajo
- Geolocalización de recursos: Asigna las visitas dinámicamente a la sala física donde se encuentra una cama disponible, manteniendo coherencia en el registro y la ubicación física del paciente.

Nodo esclavo

Arquitectura de nodos homogéneos

Físicamente, todos los nodos del clúster ejecutan exactamente el mismo código base (main.py). La distinción entre el maestro y el esclavo no es estática ni definida por hardware, sino que es un rol dinámico determinado en tiempo de ejecución. Un nodo esclavo se define como una instancia que tras un proceso de elección no alcanzó el liderazgo. Su principal función cambia de ser un orquestador a realizar replica pasiva y monitor constante.

El componente central para el nodo esclavo el StorageService, un servidor TCP multihilo diseñado para operar de manera silenciosa y eficiente. El servicio se mantiene a la escucha en un puerto dedicado, separado del puerto de gestión del nodo maestro. Esto aísla el tráfico de replicación del tráfico de clientes.

Al recibir una instrucción de escritura proveniente del maestro realiza el siguiente flujo de trabajo:

1. Deserialización: Decodifica el mensaje JSON y extrae la sentencia SQL parametrizada
2. Ejecución local: Aplica la operación directamente sobre su archivo .db
3. Confirmación (ACK): Retorna un estado de éxito o error al maestro.

Integridad y persistencia local

Cada nodo esclavo mantiene su propia copia física de la base de datos. Si el nodo maestro sufre un fallo en su disco, los nodos esclavos garantizan la persistencia de los datos. Aunque son replicas, el motor SQLite sigue imponiendo restricciones de integridad FOREIGN KEY. Esto actúa como una segunda capa de seguridad: si el maestro intenta replicar una visita para un paciente que no existe en el esclavo, este rechaza la operación y alerta al sistema de una inconsistencia.

Rol en la tolerancia a fallos

El nodo esclavo no es un elemento pasivo en el sistema que solo recibe ordenes, juega un papel activo en la resiliencia del sistema a través del módulo DetectorFallas.

- Monitoreo heartBeat: Ejecuta un hilo en segundo plano que monitorea constantemente el estado activo o inactivo de otros nodos, especialmente del maestro.
- Detección de ausencia: Utiliza un algoritmo de timeout y si el maestro deja de responder a los pings cierra su conexión. El esclavo declara la muerte del líder.
- Activación de protocolo de elección: Al confirmar la caída del nodo maestro, el esclavo deja su rol pasivo e inicia el algoritmo Bully para elegir un nuevo nodo maestro.

Interacción con el cliente

La capa de presentación fue implementada para que tuviera una interacción directa con el usuario y aislada de lógica de negocio, almacenamiento de datos o cálculos complejos. Su única responsabilidad es:

1. Capturar la entrada del usuario
2. Serializar la solicitud a formato JSON
3. Transmitir la solicitud al nodo maestro actual
4. Mostrar una respuesta recibida.

Esta arquitectura permite que el cliente sea completamente desacoplado, puede cerrarse, reiniciarse o ejecutarse en múltiples terminales simultáneamente sin afectar el estado del sistema distribuido.

Mecanismo tolerante a fallos

Uno de los desafíos principales en el sistema es que el nodo maestro puede cambiar, por lo que la dirección IP puede cambiar en cualquier momento. Para

resolver esto sin intervención manual del usuario, se implementó una lógica de descubrimiento y conmutación por error en el cliente:

- Lista de candidatos: El cliente mantiene una lista con las direcciones y puertos de todos los nodos posibles del clúster
- Algoritmo de búsqueda (Round-Robin): Al intentar enviar una petición se realiza el siguiente flujo:
    - Intenta conectar con el primer nodo de la lista
    - Si la conexión falla se asume que el nodo está caído o que no se trata del nodo maestro
    - Itera automáticamente al siguiente nodo en la lista
    - Repite el proceso hasta encontrar al maestro activo o terminar la lista.

Diseño de la interacción

Se desarrolló una interfaz de línea de comandos interactiva con el objetivo de alcanzar un entorno operativo hospitalario eficiente. Se incluyen distintos elementos que convierten a este diseño en un diseño efectivo e intuitivo:

- Menú cíclico
    - Registro de pacientes
    - Asignación de visitas
    - Monitoreo de camas disponibles por salas
    - Generación de reportes
    - Alta médica por un doctor
- Visualización de datos tabulares
- Manejo de tiempos de espera

Resultados

A lo largo del desarrollo se realizaron pruebas en dos entornos diferentes para comprobar la calidad del resultado tanto en la etapa de desarrollo como en el despliegue final.

El entorno local conserva algunas configuraciones distintas a las colocadas en el despliegue, principalmente en las IPs, manejo de terminales y manejo de scripts de prueba y depuración. Esto se reporta con el objetivo de observar las distintas etapas de desarrollo mediante los resultados obtenidos progresivamente.

Pruebas locales

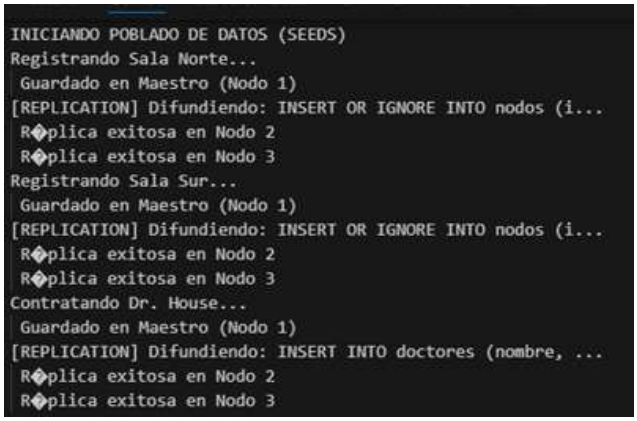

Figura 1: Ejecución de script de poblado automático de la base de datos en entorno local

Figura 2: Resultados de actualizaciones en las bases de datos con script auxiliar

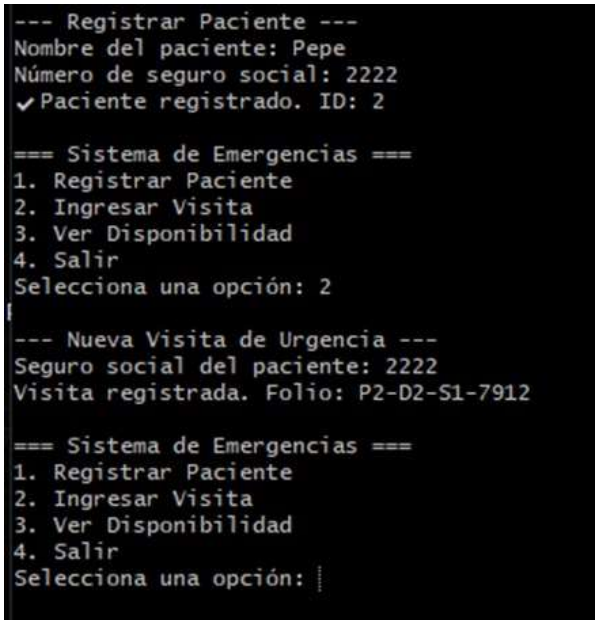

Figura 3: Obtención de la funcionalidad completa de registro de visitas en entorno local

Despliegue final

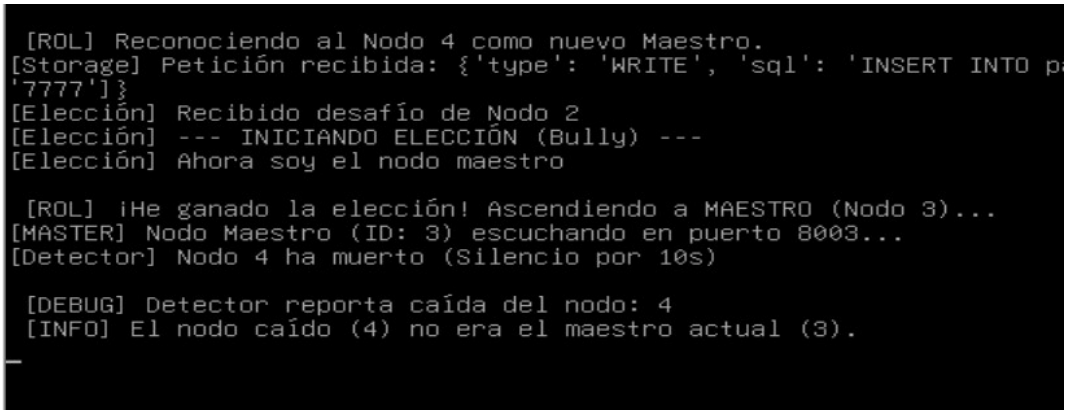

Figura 4: Log de proceso de elección de nuevo líder en despliegue final

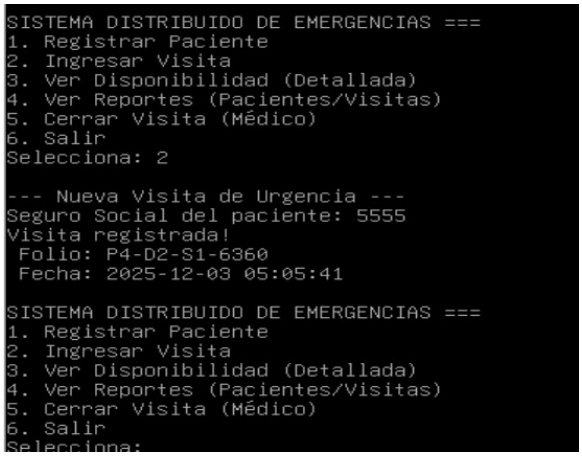

Figura 5: Generación de visita en el despliegue final

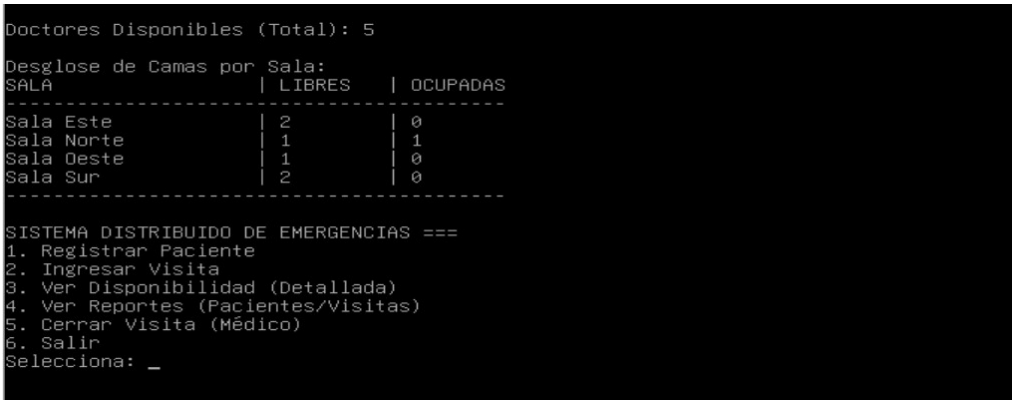

Figura 6: Disponibilidad de salas en entorno desplegado

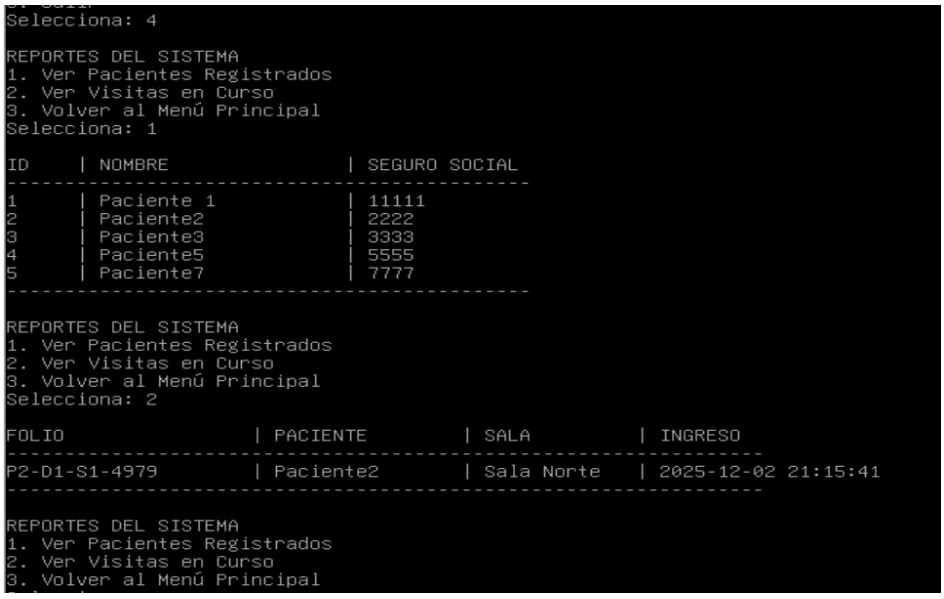

Figura 7: Reportes generados por medio de la interfaz

Para profundizar aún más sobre el funcionamiento de usuario se recomienda ver el documento adjunto “Manual de usuario.pdf”.

Conclusiones

- Juárez Andrade Axel Yael

  El desarrollo de este sistema distribuido evidenció la complejidad crítica de gestionar la concurrencia en recursos compartidos. Durante las pruebas de estrés, pude comprobar que la exclusión mutua no es solo un concepto teórico, sino una necesidad práctica para evitar la corrupción de datos (como la sobreasignación de camas). Uno de los hallazgos más importantes fue el comportamiento de 'Split-Brain' al reconectar el nodo maestro original; esto me enseñó que los algoritmos de recuperación y reconciliación de estado son tan vitales como los de elección de líder. La práctica demostró que la consistencia estricta impone latencia, pero es indispensable en sistemas críticos de salud.

- Molina Véjar Aarón Gael

  El desarrollo de un sistema distribuido completo, involucrando fases de desarrollo fundamentales como el diseño de la base de datos, la implementación de algoritmos y conceptos necesarios para el funcionamiento y cumplimiento de los requisitos no hizo más que afianzar todos los conocimientos adquiridos a lo largo del curso. Las distintas capas y etapas que fueron abordadas a lo largo del desarrollo e implementación de este proyecto resultaron sumamente desafiantes, pero también reveladores en cuanto a la aplicación de los fundamentos de los sistemas embebidos, acompañados de una gran base sólida respecto a bibliografía, documentación y ejemplos de sistemas implementados.

  Al final cada implementación tiene un motivo y una funcionalidad que provoca la consistencia del sistema completo. Analizar, organizar e implementar

fueron algunas de las habilidades esenciales en la ejecución en conjunto de este proyecto.

- Sanchez Gachuz Jennyfer Estefanya

  Este proyecto me permitió profundizar en la arquitectura Primary-Backup y su impacto en la integridad de los datos. Al implementar la replicación de SQLite a través de la red, comprendí la diferencia fundamental entre la disponibilidad de lectura (que logramos hacer local y rápida) y la consistencia de escritura (que requiere coordinación global). Observé que, aunque el sistema es tolerante a fallos parciales, la sincronización inicial de los nodos (mediante seeds) es el punto más vulnerable. Esto resalta la importancia de diseñar mecanismos de 'bootstrap' robustos para garantizar que todos los nodos inicien con una base de datos homogénea.

- Zuñiga Mosco Rodrigo

  Trabajar directamente con Sockets TCP y la serialización de mensajes JSON me brindó una perspectiva clara sobre la capa de transporte. El reto de implementar el framing para evitar la fragmentación de paquetes en el flujo TCP fue crucial para la estabilidad del cliente. Además, contrastar la teoría de los algoritmos de consenso con su implementación en Python reveló las dificultades de mantener el estado global sincronizado sin un reloj común. Este proyecto integró exitosamente los conceptos de transparencia y escalabilidad, demostrando cómo una arquitectura modular facilita el mantenimiento y la expansión futura del clúster.

Referencias

[1] Tanenbaum, A. S. (2011). *Redes de computadoras* (5ª ed.). Prentice Hall. (Trabajo original publicado en inglés en 2010).

[2] Silberschatz, A., Korth, H. F. y Sudarshan, S. (2006). *Fundamentos de bases de datos* (5ª ed.). McGraw-Hill. (Trabajo original publicado en inglés en 2005).

[3] Coulouris, G., Dollimore, J., Kindberg, T. y Blair, G. (2012). *Distributed systems: Concepts and design* (5ª ed.). Addison-Wesley.

[4] Tanenbaum, A. S. y van Steen, M. (2007). *Distributed systems: Principles and paradigms* (2ª ed.). Prentice Hall.

[5] SQLite. (2023). *SQLite documentation* (Versión 3.45.0) [Documentación de software]. https://sqlite.org/docs.html