El sistema a desarrollar tiene como objetivo principal garantizar la confiabilidad, seguridad, escalabilidad y alta disponibilidad para la consulta de información de votantes durante el proceso electoral. Esto es fundamental debido a la naturaleza crítica y masiva del evento, que involucra a más de 100 millones de votantes distribuidos en 1000 ciudades. Para abordar los requerimientos, se ha diseñado una arquitectura basada en capas que separa responsabilidades y protege la base de datos de accesos directos por parte de los clientes.

Dado el contexto, la estrategia del sistema combina principios arquitectónicos sólidos y patrones de diseño específicos que permiten abordar los retos del sistema de manera eficiente y modular. La solución se centra en lo siguiente:

* Comunicación eficiente y segura: Asegurar que los clientes interactúen únicamente con la capa intermedia (servidor) mediante un protocolo definido (ICE), protegiendo la base de datos y garantizando integridad.
* Distribución de carga y concurrencia: Diseñar mecanismos para manejar grandes volúmenes de consultas concurrentes, asegurando tiempos de respuesta dentro del límite definido.
* Escalabilidad y flexibilidad: Permitir la incorporación dinámica de más clientes y el manejo eficiente de variaciones en la carga de trabajo.
* Auditoría y trazabilidad: Registrar la información generada durante las consultas tanto en el servidor como en los clientes, garantizando que el sistema sea auditable y verificable.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Patrón | Propósito | Ventajas | Desventajas | Complejidad de Implementación | Coste de Implementación | Relevancia para el Problema |
| **Broker** | Coordina la comunicación entre clientes y servicios distribuidos a través de una capa intermedia. | - Aísla la lógica de negocio del cliente.  - Facilita escalabilidad y modularidad.  - Mejora la seguridad. | - Mayor complejidad de implementación.  - Dependencia de un mediador central. | Alta | Alto | Esencial para manejar la comunicación distribuida y garantizar escalabilidad, reduciendo la carga de los clientes. |
| **Observer** | Permite que múltiples objetos sean notificados automáticamente cuando ocurre un cambio en otro objeto. | - Sincronización en tiempo real.  - Reducción del acoplamiento entre componentes.  - Fácil de extender. | - Puede volverse ineficiente con muchos observadores.  - Difícil de depurar. | Media | Moderado | Facilita notificaciones eficientes en tiempo real a los clientes sobre actualizaciones o resultados de consultas. |
| **Master-Worker** | Divide las tareas en subtareas y las distribuye entre varios workers para procesarlas en paralelo. | - Mejora el rendimiento y la concurrencia.  - Distribución eficiente de la carga.  - Escalable. | - Sobrecarga de gestión para el master.  - Puede fallar si no se maneja bien. | Alta | Moderado-Alto | Perfecto para dividir y distribuir las consultas entre múltiples nodos, manejando grandes volúmenes de trabajo sin saturar el sistema. |
| **Singleton** | Garantiza que una clase tenga una única instancia y proporciona un punto global de acceso a ella. | - Control centralizado de recursos.  - Fácil de implementar. | - Genera acoplamiento y dificulta pruebas unitarias. | Baja | Bajo | No relevante, ya que el problema requiere múltiples instancias y distribución de tareas. |
| **Factory Method** | Proporciona una interfaz para crear objetos, permitiendo que las subclases determinen qué clase instanciar. | - Flexibilidad en la creación de objetos.  - Facilita pruebas unitarias. | - Sobrecarga de código para clases adicionales. | Media | Moderado | Poco relevante, ya que el enfoque está más orientado al procesamiento y distribución de tareas que a la creación dinámica de objetos. |
| **Facade** | Proporciona una interfaz simplificada para un conjunto de interfaces más complejas. | - Reduce la complejidad percibida por los clientes.  - Mejora la mantenibilidad. | - No resuelve problemas de escalabilidad ni concurrencia. | Baja | Bajo | Ayuda a simplificar la interacción cliente-servidor, pero no resuelve problemas de distribución de carga como lo hace el Broker. |
| **Proxy** | Proporciona un sustituto o representante para controlar el acceso a otro objeto. | - Mejora el control de acceso.  - Reduce carga inicial. | - Añade latencia.  - Dificultad para mantener. | Media | Moderado | Aunque es útil para controlar accesos, no es crítico para el problema descrito. |
| **Decorator** | Agrega responsabilidades dinámicamente a los objetos. | - Flexible para extender funcionalidad.  - No afecta otras instancias. | - Puede complicar el diseño.  - Genera muchas clases adicionales. | Media | Moderado | No aplica directamente, ya que el sistema no necesita añadir dinámicamente funcionalidades a los objetos. |
| **Composite** | Compone objetos en estructuras de árbol para tratarlos de forma uniforme. | - Manejo jerárquico de componentes.  - Simplifica el diseño. | - Sobrecarga en objetos simples.  - Mayor consumo de memoria. | Media | Moderado | Poco relevante, ya que no hay jerarquías complejas en el problema que resolver. |
| **Chain of Responsibility** | Permite que una solicitud pase a través de una cadena de objetos para ser procesada. | - Flexible para añadir manejadores.  - Reduce el acoplamiento. | - Dificultad para depurar.  - Sin garantía de que la solicitud sea manejada. | Media | Moderado | Puede ser útil para procesar ciertas consultas, pero no cubre eficientemente los problemas de escalabilidad y distribución como los patrones seleccionados. |
| **Adapter** | Permite que clases incompatibles trabajen juntas al proporcionar una interfaz común. | - Facilita la integración de sistemas existentes.  - Reutilización de código. | - Puede añadir complejidad innecesaria.  - No optimiza el rendimiento. | Baja | Bajo | No aplica directamente, ya que el problema no implica compatibilidad entre interfaces de sistemas existentes. |
|  |  |  |  |  |  |  |

Para cumplir con estos objetivos, la estrategia implementa tres patrones clave: Observer, Master-Worker y Broker. Estos patrones no solo permiten satisfacer los requerimientos funcionales y no funcionales, sino que también aseguran que el sistema pueda evolucionar y adaptarse a futuros escenarios.

1. **Patrón Observer**

Descripción:

Este patrón define una relación de dependencia uno-a-muchos entre objetos. Cuando el estado de un objeto (el "observado") cambia, todos sus "observadores" son notificados automáticamente.

**Aplicación en el Sistema:**

**Requerimiento relevante: E7.**

* El servidor actúa como el observado.
* Los clientes actúan como observadores que se registran en el servidor para recibir actualizaciones.

**Problemas que resuelve:**

* Sincronización eficiente: Permite mantener actualizados a los clientes con los cambios en el servidor, garantizando consistencia en tiempo real.
* Escalabilidad: Permite añadir nuevos clientes sin modificar la lógica principal del servidor.

**Satisfacción de necesidades:**

* Alta disponibilidad y escalabilidad (E3): Los clientes pueden registrarse dinámicamente y recibir notificaciones sin afectar el rendimiento del servidor.
* Auditoría y logging (E10): Facilita que los clientes registren información precisa en sus logs al recibir datos directamente del servidor.

Con base en el enunciado del proyecto, los patrones de diseño Observer, Master-Worker y Broker pueden explicarse y justificarse como parte de la estrategia para abordar los requerimientos funcionales y de calidad del sistema. A continuación, detallo cada patrón y su contribución al sistema:

**1. Patrón Observer**

**Descripción:**

Este patrón define una relación de dependencia uno-a-muchos entre objetos. Cuando el estado de un objeto (el "observado") cambia, todos sus "observadores" son notificados automáticamente.

**Aplicación en el Sistema:**

* **Requerimiento relevante:** E7.
  + El servidor actúa como el observado.
  + Los clientes actúan como observadores que se registran en el servidor para recibir actualizaciones.

**Problemas que resuelve:**

* Sincronización eficiente: Permite mantener actualizados a los clientes con los cambios en el servidor, garantizando consistencia en tiempo real.
* Escalabilidad: Permite añadir nuevos clientes sin modificar la lógica principal del servidor.

**Satisfacción de necesidades:**

* **Alta disponibilidad y escalabilidad (E3):** Los clientes pueden registrarse dinámicamente y recibir notificaciones sin afectar el rendimiento del servidor.
* **Auditoría y logging (E10):** Facilita que los clientes registren información precisa en sus logs al recibir datos directamente del servidor.

**2. Patrón Master-Worker**

**Descripción:**

Este patrón divide una tarea en subtareas distribuidas entre "workers" (trabajadores) que las ejecutan de manera concurrente, mientras un "master" coordina la asignación de estas tareas.

**Aplicación en el Sistema:**

* **Requerimiento relevante:** E8.
  + El servidor distribuye la lista de documentos de identidad a los clientes registrados para procesarlos en paralelo.
  + Cada cliente realiza las consultas asignadas de manera independiente.

**Problemas que resuelve:**

* Balance de carga: Divide el procesamiento entre varios clientes, evitando cuellos de botella en el servidor.
* Concurrencia: Facilita el manejo eficiente de múltiples consultas simultáneas.

**Satisfacción de necesidades:**

* **Baja latencia (E5):** La paralelización asegura que las consultas se realicen en el menor tiempo posible.
* **Escalabilidad (E3):** Se pueden añadir más clientes para manejar cargas crecientes.

**3. Patrón Broker**

**Descripción:**

El patrón Broker organiza la comunicación entre componentes distribuidos a través de un intermediario (broker), que enruta las solicitudes de un cliente a un proveedor de servicios.

**Aplicación en el Sistema:**

* **Requerimientos relevantes:** E4, E5.
  + ICE actúa como el broker, gestionando las comunicaciones entre clientes y servidor.
  + Los clientes se comunican con el broker, quien a su vez se conecta con el servidor para consultar la base de datos.

**Problemas que resuelve:**

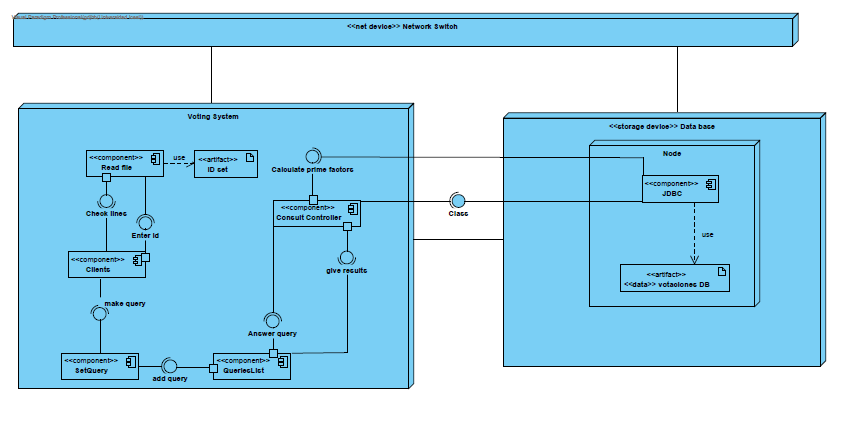
* Abstracción de comunicación: Los clientes no necesitan conocer los detalles internos del servidor o la base de datos.
* Seguridad: Garantiza que los clientes no accedan directamente a la base de datos.

**Satisfacción de necesidades:**

* **Seguridad (E4):** Evita accesos directos a la base de datos.
* **Escalabilidad y disponibilidad (E3):** Facilita la gestión de múltiples clientes y consultas de manera eficiente.

**Propuesta de solución**

**Monolito**



Distribuido:

