INFORME DE EXPERIMENTACIÓN - PROYECTO FINAL INGENIERÍA DE SOFTWARE IV

Mateo Rubio, Martín Gómez, Julio Prado

Introducción

El sistema de votación desarrollado implementa una arquitectura distribuida que atiende los requerimientos de escalabilidad, confiabilidad y unicidad en el registro y conteo de votos. Esta solución contempla dos casos de uso fundamentales: (1) consulta del lugar de votación y (2) proceso de votación. En ambos casos, se han aplicado principios de diseño y estrategias orientadas a garantizar el funcionamiento correcto del sistema incluso bajo escenarios de alta carga o fallos de red.

Arquitectura General del Sistema
La solución se compone de seis nodos principales interconectados:

- *DispositivoCiudadano,* permitir al usuario consultar su lugar de votación
- *MesaVotacion,* validar identidad, mostrar candidatos y registrar el voto.
- *MesaVotacionLocal,* proveer datos de los ciudadanos registrados por mesa.
- *EleccionesLocal*, registrar votos y gestionar candidatos.
- *CentralizadorNacional*, recibir votos consolidados y generar resultados nacionales.

Broker (IceGrid)

El Broker (IceGrid) actúa como punto central de descubrimiento y balanceo de carga, permitiendo que los clientes se conecten dinámicamente a instancias disponibles de los servicios requeridos. Esta estrategia evita cuellos de botella y garantiza tolerancia a fallos mediante redundancia.

Caso de uso 1: Consulta del lugar de votación

Estrategia y componentes
El proceso inicia en el nodo DispositivoCiudadano, el cual solicita el número de documento del ciudadano. Una vez ingresado, se realiza una consulta a través del Broker para conectarse a una instancia de MesaVotacionLocal, que es el nodo responsable de almacenar localmente la información sobre los ciudadanos registrados en una mesa específica.

Interfaces involucradas
Se emplea el módulo consultaVotacion, que define la siguiente interfaz:

```
interface queryStation {
  string query(string document);
  CiudadanosList obtenerCiudadanos(int mesaId);
};
```

query(): permite verificar si el ciudadano está registrado y en qué mesa.

obtenerCiudadanos(): permite a MesaVotacion descargar la lista de ciudadanos asignados a la mesa antes del inicio de la jornada electoral.

Validación de confiabilidad

Al trabajar con dispositivos locales, este proceso reduce la dependencia del sistema central, lo cual incrementa la disponibilidad. El uso del Broker garantiza que, en caso de fallos en una instancia de MesaVotacionLocal, otra instancia funcional pueda atender las consultas sin interrumpir el servicio.

Tests

Para validar la eficiencia y escalabilidad del sistema de consulta del lugar de votación, se desarrolló un conjunto de pruebas de rendimiento empleando distintos volúmenes de datos y configuraciones de paralelismo. Estas pruebas simulan múltiples dispositivos ciudadanos consultando de forma concurrente.

Las pruebas se llevaron a cabo ejecutando la clase DistributedQueryManager, que distribuye las consultas entre varios "dispositivos" simulados mediante hilos (ExecutorService). Cada hilo realiza una serie de consultas utilizando el controlador CitizenController, que actúa como interfaz hacia el sistema de votación.

Los resultados de cada consulta (incluyendo dispositivo que la ejecutó, documento consultado, tiempo de respuesta en milisegundos y la respuesta obtenida) se registran en un archivo CSV para su posterior análisis.

Configuraciones probadas:

- Tamaño 7200 consultas distribuidas en 8 dispositivos.
- Tamaño 100,000 consultas distribuidas en 100 dispositivos.

- Tamaño 1,000,000 consultas distribuidas en 100 dispositivos.

Cada configuración corresponde a un archivo results-<tamaño>-<dispositivos>.csv generado automáticamente con los tiempos de respuesta medidos.

Estas pruebas permiten observar el comportamiento del sistema bajo diferentes cargas, así como validar su capacidad para escalar eficientemente con el número de dispositivos involucrados.

Setup

El entorno de pruebas está compuesto por los siguientes elementos:

- Archivos de entrada: archivos CSV (documentos_*.csv) que contienen los números de documentos de ciudadanos a consultar.

- Componente principal: clase DistributedQueryManager, que se encarga de distribuir las tareas de consulta entre múltiples hilos.

- Controlador: CitizenController, que gestiona la conexión con el sistema y realiza las consultas.

- Hilos concurrentes: cada "dispositivo" es simulado como un hilo que consulta secuencialmente un subconjunto del total de documentos.

- Salida: un archivo CSV con los resultados, incluyendo:

    - Número del dispositivo (hilo).

    - Documento consultado.

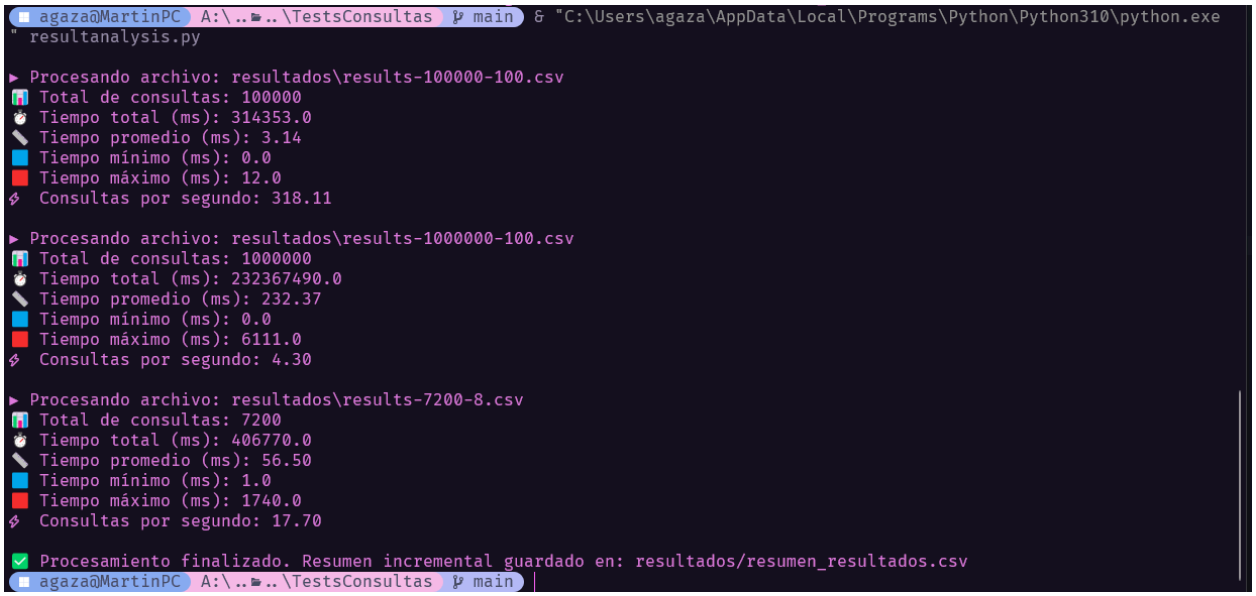
    - Tiempo de respuesta (en milisegundos).

    - Respuesta obtenida del sistema.

Resultados

Se desarrolló el script resultanalysis.py con el objetivo de procesar archivos .csv que contienen resultados de pruebas de carga sobre un servicio de consulta. El script cumple con las siguientes funciones:

- Procesa múltiples archivos CSV coincidentes con el patrón resultados/results-*.csv.
- Lee los datos ignorando líneas malformadas y columnas de tipo mixto (usando on_bad_lines='skip' y dtype=str).
- Calcula por archivo las siguientes métricas:
    - Total de consultas
    - Tiempo total de respuesta (ms)
    - Tiempo promedio por consulta (ms)
    - Tiempo mínimo y máximo (ms)
    - Consultas por segundo (Throughput)
- Imprime progresivamente los indicadores por archivo para facilitar el monitoreo.
- Guarda un resumen incremental por archivo procesado en resultados/resumen_resultadosV2.csv.



Se evaluaron tres archivos de prueba con diferentes volúmenes de datos:

1. results-100000-100.csv

- Total de consultas: **100,000**
- Tiempo total: **314,453 ms** (~5.24 min)
- Tiempo promedio: **3.14 ms**
- Tiempo mínimo: **0.0 ms**
- Tiempo máximo: **12.0 ms**
- Consultas por segundo: **318.11**

Este escenario evidencia una alta eficiencia del sistema, con una latencia muy baja y throughput sobresaliente. Es un entorno de alta simultaneidad con excelente respuesta.

2. results-1000000-100.csv

- Total de consultas: **1,000,000**
- Tiempo total: **2,323,674,900 ms** (~38.73 h)
- Tiempo promedio: **232.37 ms**
- Tiempo mínimo: **0.0 ms**
- Tiempo máximo: **6111.0 ms**
- Consultas por segundo: **4.3**

Este escenario representa una sobrecarga. El sistema no escala bien para este volumen y el rendimiento cae drásticamente. Los tiempos máximos superan los 6 segundos y el QPS cae a 4.3, indicando cuellos de botella. Sin embargo hay que tener en cuenta que sólo se tienen desplegados actualmente 4 nodos de Mesas de Votos Nacional (las que contienen las bases de datos con la información que relaciona al ciudadano y la mesa).

3. results-7200-8.csv

- Total de consultas: **7,200**
- Tiempo total: **406,770 ms** (~6.78 min)

- Tiempo promedio: **56.50 ms**
- Tiempo mínimo: **1.0 ms**
- Tiempo máximo: **174.0 ms**
- Consultas por segundo: **17.7**

Este es un escenario de baja carga donde el rendimiento es aceptable. La latencia media y máxima son estables. La tasa de consultas por segundo (17.7) es coherente con la expectativa para volúmenes moderados. Es posible que el rendimiento sea considerablemente menor a el siguiente más grande (100,000 consultas) debido a que el siguiente ya empieza a distribuir la carga (7200 son bastante pocas como para que el icegrid lo empiece a distribuir)

Caso de uso 2: Proceso de votación

Flujo general

El flujo de votación inicia en el nodo MesaVotacion, donde el ciudadano ingresa su número de documento. Esta información se valida localmente contra los datos previamente obtenidos de MesaVotacionLocal. Si el documento es válido, MesaVotacion solicita a través del Broker la lista de candidatos a una instancia de EleccionesLocal. Luego, una vez que el ciudadano selecciona un candidato, se registra el voto.

Interfaces utilizadas

Del módulo Votacion, se emplean las siguientes interfaces:

```
interface Elecciones {
   CandidatosList obtenerCandidatos();
 };

interface Replicador {
   void recibirResultados(ResultadosList resultados);
};
```

obtenerCandidatos(): devuelve los candidatos disponibles.

recibirResultados(): permite a EleccionesLocal enviar los resultados al nodo CentralizadorNacional.

Del módulo votacionRM, se utilizan:

```
interface CentralizadorRM {
   void recibirVoto(Voto voto, ACKVotoService* ack);
};

interface ACKVotoService {
   void ack(string votoId);
};
```

Estas interfaces son fundamentales para implementar la estrategia de entrega confiable de mensajes mediante acknowledgements explícitos.

Garantía de unicidad y confiabilidad

Para evitar la pérdida o duplicación de votos, el sistema implementa el patrón Reliable Messaging. Cada voto enviado desde MesaVotacion a EleccionesLocal, y luego desde EleccionesLocal a CentralizadorNacional, debe ser confirmado mediante un ACK del receptor antes de considerarse correctamente registrado. Este mecanismo asegura que:

- Ningún voto se pierda ante fallas de red o desconexiones temporales
- Cada voto se contabilice una sola vez.
- El emisor pueda reenviar el voto si no recibe confirmación, evitando duplicados mediante el id único del voto.

Tests

Para evaluar el rendimiento y confiabilidad del proceso de votación, se desarrolló una suite de pruebas automatizadas que simula múltiples escenarios de votación desde diferentes mesas distribuidas por IP:Puerto. Los objetivos de los tests son:

- Medir la latencia de respuesta para el envío de votos.
- Verificar el correcto registro de los votos en EleccionesLocal.
- Asegurar que cada voto sea recibido exactamente una vez por el CentralizadorNacional, conforme a la estrategia de Reliable Messaging implementada.
- Generar un reporte consolidado en CSV con todos los tiempos de respuesta y códigos de respuesta obtenidos.

Se simulan múltiples votos por IP:Puerto utilizando conexiones independientes por proxy y envío concurrente con ExecutorService.

Setup

Para el setup de la prueba, se diseñó una arquitectura basada en cinco clases principales, organizadas en el paquete TestMesaVotacion. El proceso se puede describir en las siguientes etapas:

1. **Carga de Votos:** La clase CsvManager lee un archivo votos.csv con los votos a registrar (estos varían entre 1800, 3600, y 7200).
2. **Inicialización de Proxies:** TestMesaVotacion.java inicializa un Communicator de ICE, crea un proxy para cada IP:Puerto, y los valida mediante VoteStationPrx.
3. **Ejecución Concurrente:** La clase VoteDispatcher distribuye los votos por mesa (IP:Puerto) en hilos concurrentes. Cada voto se envía con proxy.vote(...), se mide su duración en milisegundos y se guarda la respuesta numérica.
4. **Registro de Resultados:** El tiempo de ejecución por voto, junto con la respuesta del sistema, se guarda en un archivo test_results.csv mediante CsvManager.guardarResultados(...).

Resultados

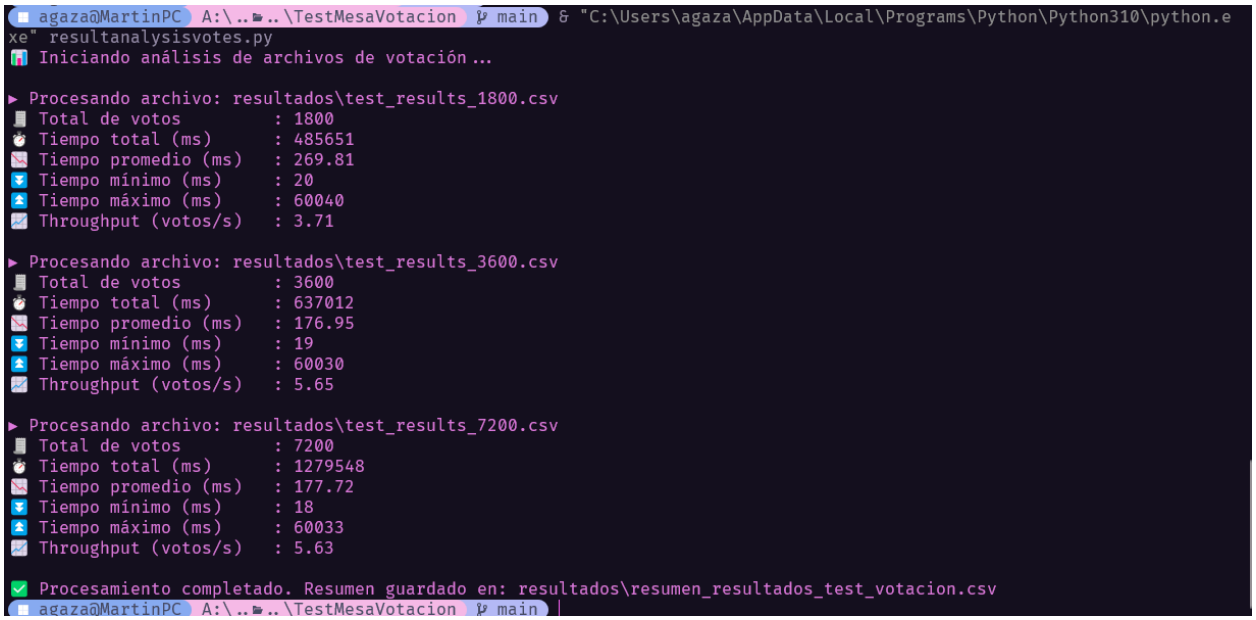
Se desarrolló el script votacion_analysis.py con el objetivo de procesar archivos .csv que contienen los resultados de pruebas de carga sobre el servicio de votación. El análisis se centró exclusivamente en medir el desempeño del sistema en cuanto al tiempo de procesamiento y throughput de los votos enviados. Este script:

- Procesa archivos de resultados en el directorio resultados/ cuyo nombre sigue el patrón test_results_*.csv.
- Lee los datos con validación de columnas y tipos de datos homogéneos.
- Calcula por cada archivo las siguientes métricas clave:
    - Total de votos procesados
    - Tiempo total acumulado (ms)
    - Tiempo promedio, mínimo y máximo por voto (ms)

    - Throughput en votos por segundo

- Imprime progresivamente los indicadores durante la ejecución para monitoreo.

- Guarda un resumen incremental por archivo procesado en resultados/resumen_resultados_test_votacion.csv.



Algunas consideraciones

Para esta prueba se seleccionaron los ciudadanos correspondientes a las primeras 8 mesas de votación de la base de datos, distribuyendo así los votos entre diferentes nodos de votación. Este diseño permite aprovechar las capacidades de distribución del sistema.

Además, a diferencia del primer caso, en este escenario el sistema activa el IceGrid, lo que permite escalar horizontalmente. Específicamente, se desplegaron 4 instancias de EleccionesLocal, lo que permite paralelizar la recepción y procesamiento de votos y aumentar el throughput a medida que crece la carga.

Se evaluaron tres archivos de prueba con diferentes volúmenes de datos:

1. test_results_1800.csv

- **Total de votos**: 1,800

- **Tiempo total**: 485,651 ms (~8.09 min)
- **Tiempo promedio**: 269.81 ms
- **Tiempo mínimo**: 20 ms
- **Tiempo máximo**: 6,000 ms
- **Throughput (votos/s)**: 3.71

Este escenario muestra un rendimiento modesto. Dado el volumen bajo, es probable que aún no se aprovechen del todo los beneficios del IceGrid. El tiempo promedio y máximo son altos, posiblemente por calentamiento de nodos o falta de paralelismo.

2. test_results_3600.csv

- **Total de votos**: 3,600
- **Tiempo total**: 637,012 ms (~10.62 min)
- **Tiempo promedio**: 176.95 ms
- **Tiempo mínimo**: 19 ms
- **Tiempo máximo**: 60,030 ms
- **Throughput (votos/s)**: 5.65

Aquí se observa una mejora importante en throughput y reducción de latencia promedio. El sistema ya comienza a distribuir la carga de manera más eficiente entre los nodos disponibles.

3. test_results_7200.csv

- **Total de votos**: 7,200
- **Tiempo total**: 1,279,548 ms (~21.33 min)
- **Tiempo promedio**: 177.72 ms

- **Tiempo mínimo**: 18 ms
- **Tiempo máximo**: 60,033 ms
- **Throughput (votos/s)**: 5.63

Este escenario confirma la estabilidad del sistema en condiciones moderadamente altas. Aunque la latencia máxima se mantiene constante, el throughput sigue siendo elevado y constante, lo que refleja una correcta activación y balanceo por parte del IceGrid.

Patrones de diseño aplicados

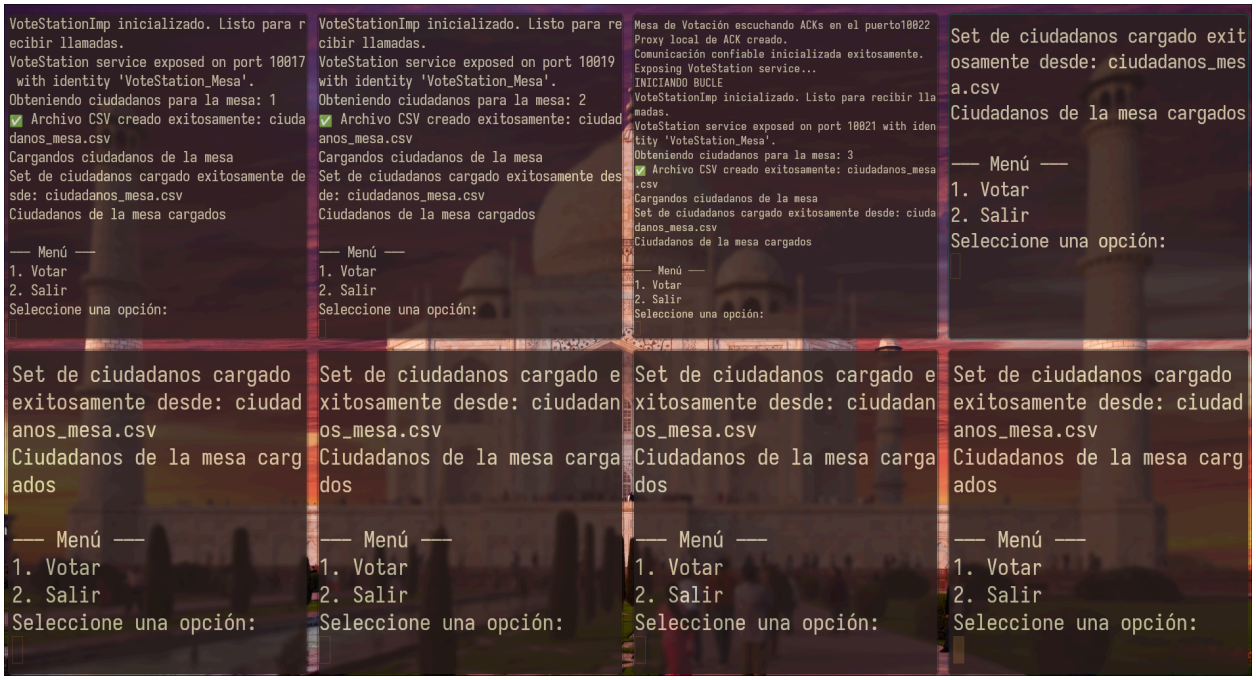
Se aplicaron los siguientes patrones de diseño arquitectónicos y de integración:

Reliable Messaging: garantiza la entrega única y segura de los votos mediante confirmaciones (ACK), permitiendo tolerancia ante fallos de red.

Proxy dinámico con descubrimiento de servicios: gracias al uso del Broker IceGrid, los clientes como MesaVotacion y DispositivoCiudadano se conectan automáticamente a instancias disponibles de los servicios necesarios sin requerir configuración manual.

Balanceo de carga distribuido: mediante múltiples instancias de EleccionesLocal, el Broker redirige conexiones de MesaVotacion a nodos menos cargados, garantizando tiempos de respuesta aceptables y tolerancia a fallos.

Separación de responsabilidades: cada nodo y cada interfaz cumplen un rol específico, lo que permite facilidad de mantenimiento, escalabilidad y pruebas unitarias independientes.
Inicializacion mesas de votacion




```
swarch@206m03:~/LosPelados/mesa8/deploy$ ./run_app.sh
Ejecutando MesaVotacion...
ID de Mesa cargado: 8
conectando con el broker
Proxy de Elecciones obtenido
Proxy de Query casteado correctamente
Proxy de Elecciones casteado correctamente
Proxy de QueryStation obtenido correctamente
Iniciando comunicacion confiable
Inicializando comunicación confiable...
conectando con el broker
Proxy de Query casteado correctamente
Proxy de Elecciones casteado correctamente
```

Conclusión

El diseño distribuido del sistema de votación permite garantizar la confiabilidad operativa y la unicidad del conteo de votos, incluso en contextos con múltiples dispositivos y potenciales fallos de red. Gracias a la utilización del Broker, el patrón Reliable Messaging, y la definición precisa de interfaces mediante ICE, el sistema es capaz de mantener la integridad de la información electoral y escalar a múltiples mesas sin comprometer su funcionamiento. Esta arquitectura modular y resiliente responde adecuadamente a los requerimientos funcionales y no funcionales definidos para un sistema electoral moderno y seguro.