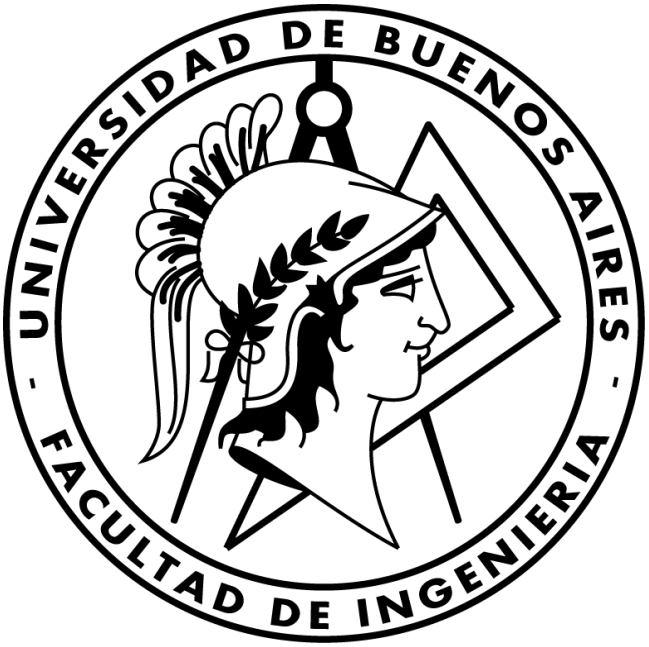


TP Tolerancia

Steam Analysis

Integrantes

Alumnos

Berenguel, Rafael - rberenguel@fi.uba.ar

Gazzola, Franco- fgazzola@fi.uba.ar

Javes, Sofia - sjaves@fi.uba.ar

Tutor

Tomas Nocetti - tnocetti@fi.uba.ar

Introducción

El presente informe tiene como objetivo detallar la solución implementada para el trabajo práctico *"Tolerancia"*, correspondiente al curso **Sistemas Distribuidos 1 (75.74)** de la carrera de Ingeniería Informática en la Universidad de Buenos Aires.

Este trabajo práctico consistió en el diseño y desarrollo de un sistema distribuido capaz de analizar reseñas de videojuegos disponibles en la plataforma *Steam*. La solución se basó en *datasets* obtenidos de la plataforma Kaggle y empleó tecnologías que permiten escalar el sistema, gestionar la comunicación entre nodos mediante un middleware y garantizar la tolerancia a fallos en las etapas finales del proyecto.

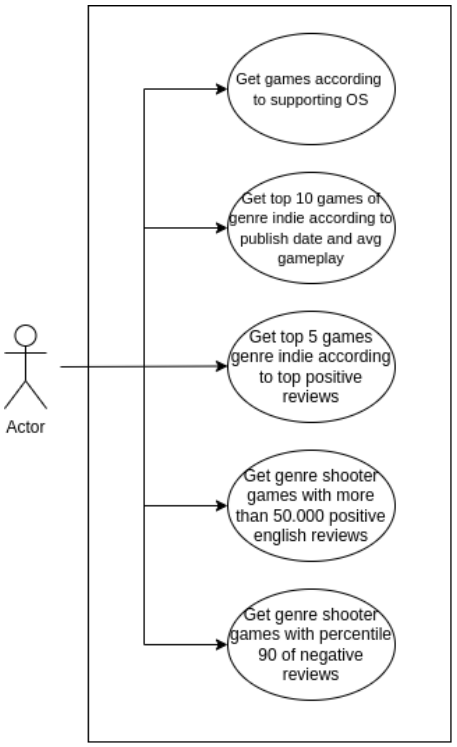
El sistema fue diseñado siguiendo los principios teóricos del curso, priorizando flexibilidad, escalabilidad y robustez en un entorno multi computadora. Asimismo, su desarrollo se estructuró en entregas progresivas, cumpliendo con requisitos específicos definidos para cada etapa.

Casos de Uso

En este caso recordemos las consultas a resolver:

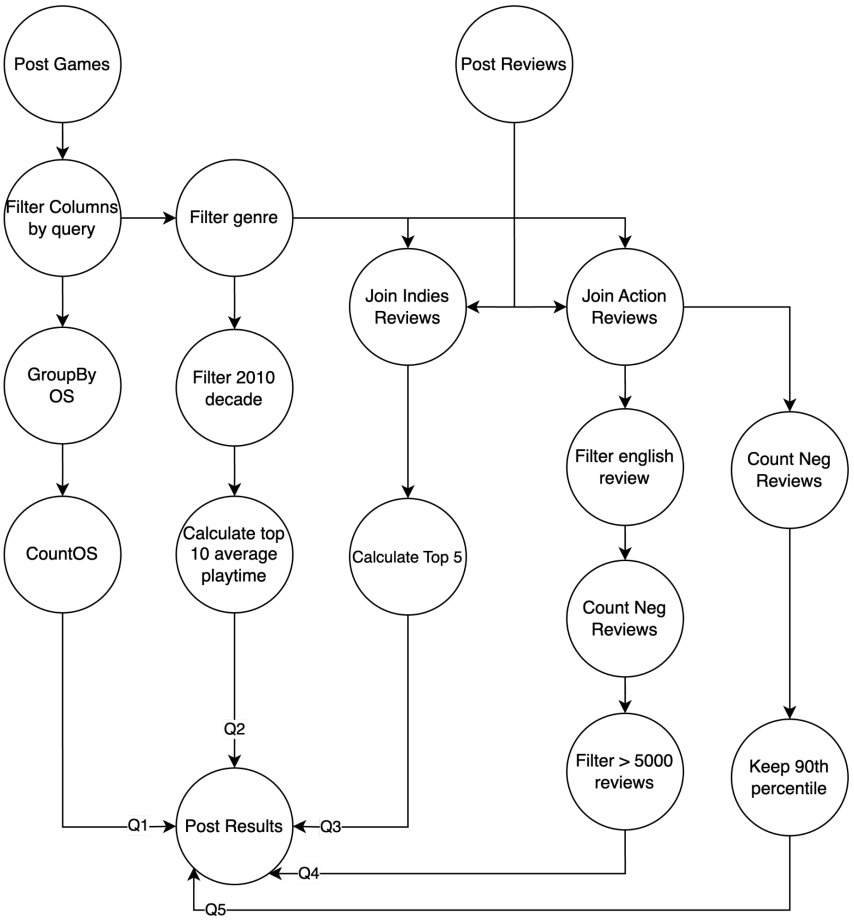
- Cantidad de juegos soportados en cada plataforma (Windows, Linux, MAC)
- Nombre de los juegos top 10 de genero Indie publicados en la década del 2010 con mas tiempo promedio histórico de juego
- Nombre de los juegos top 5 del genero Indie con más reseñas positivas
- Nombre de juegos del genero action con más de 5.000 reseñas negativas en idioma inglés
- Nombre de juegos del género action dentro del percentil 90 en cantidad de reseñas negativas

Por lo que podemos observar el siguiente diagrama de casos de uso:



Vista lógica

DAG

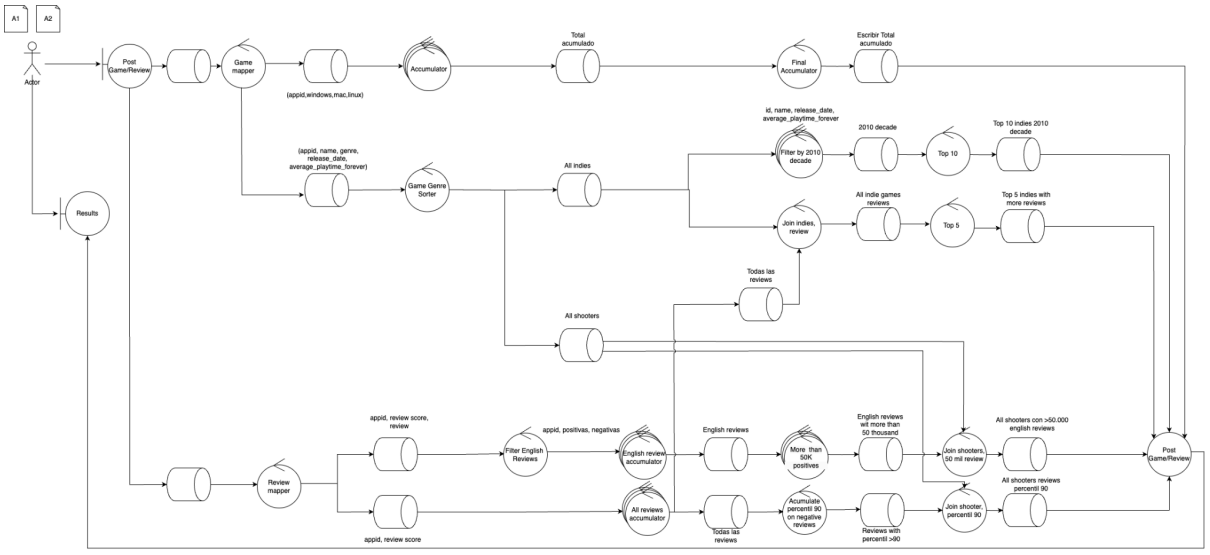


En el diagrama podemos observar la división del manejo de juegos y por otro lado las reseñas. Los juegos en primer lugar sufrirán modificaciones a tal punto de solamente mantener aquellos de género que sean de importancia para los usos que necesitemos, en este caso: Indie y action. Todos aquellos juegos que no pertenezcan a estos géneros serán descartados. Luego de esto se hace un join de cada género en particular. A partir de esto comienza el procesamiento de cada consulta en particular ya sea: un conteo por cada sistema operativo soportado, filtros de idioma, cálculo de percentil y más.

Vista Física

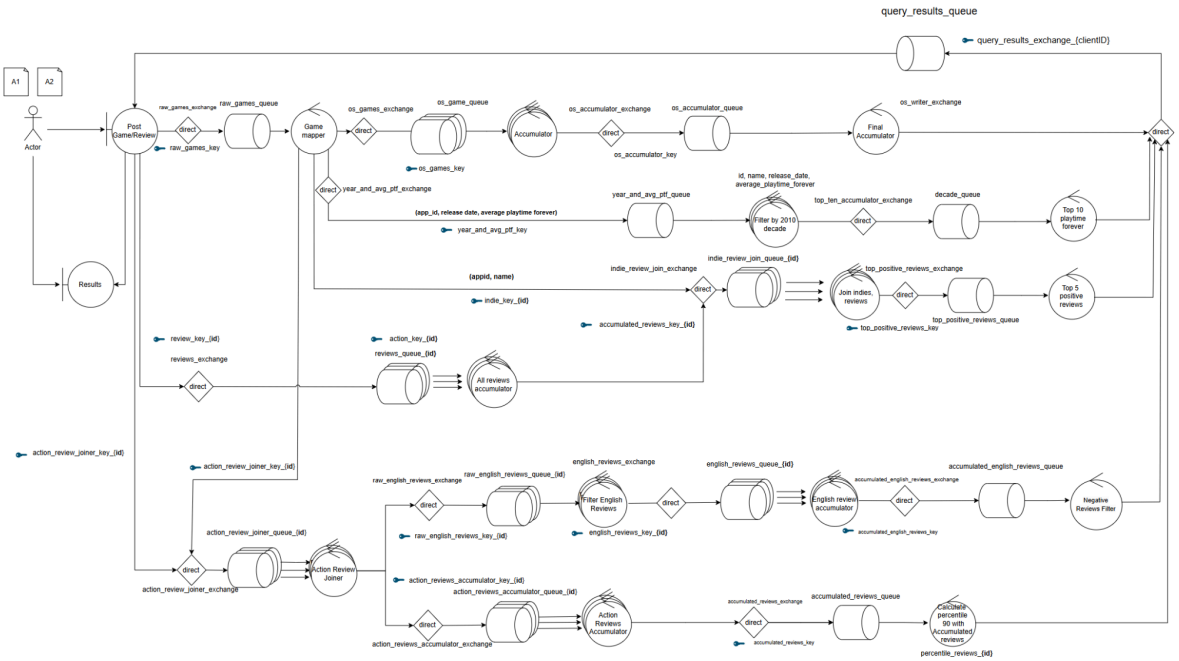
Diagrama de Robustez

Nos parece interesante remarcar las modificaciones que fue sufriendo el Trabajo a medida que avanzabamos en la distintas iteraciones hasta el dia de hoy por lo que a continuación se muestra el primer Diagrama de Robustez diseñado para el presente Trabajo:



Se observa como al inicio comenzamos con los nodos de cada query más diferenciados entre sí. Creímos también que deshacernos del texto de las reseñas era lo primero que debíamos hacer para un funcionamiento más fluido y de mejor optimización, obsérvese la query N4 (comenzando por el English Filter) el cual era el encargado de detectar aquellas reseñas con texto en idioma inglés para luego descartar esta parte. Teníamos también 2 joiners que eran muy similares tanto en funcionamiento como en manejo de mensajes.

Actualmente, como bien se dijo, este diagrama ya no es el que utilizamos para representar el flujo y funcionamiento de nuestro Trabajo como se puede apreciar a continuación:

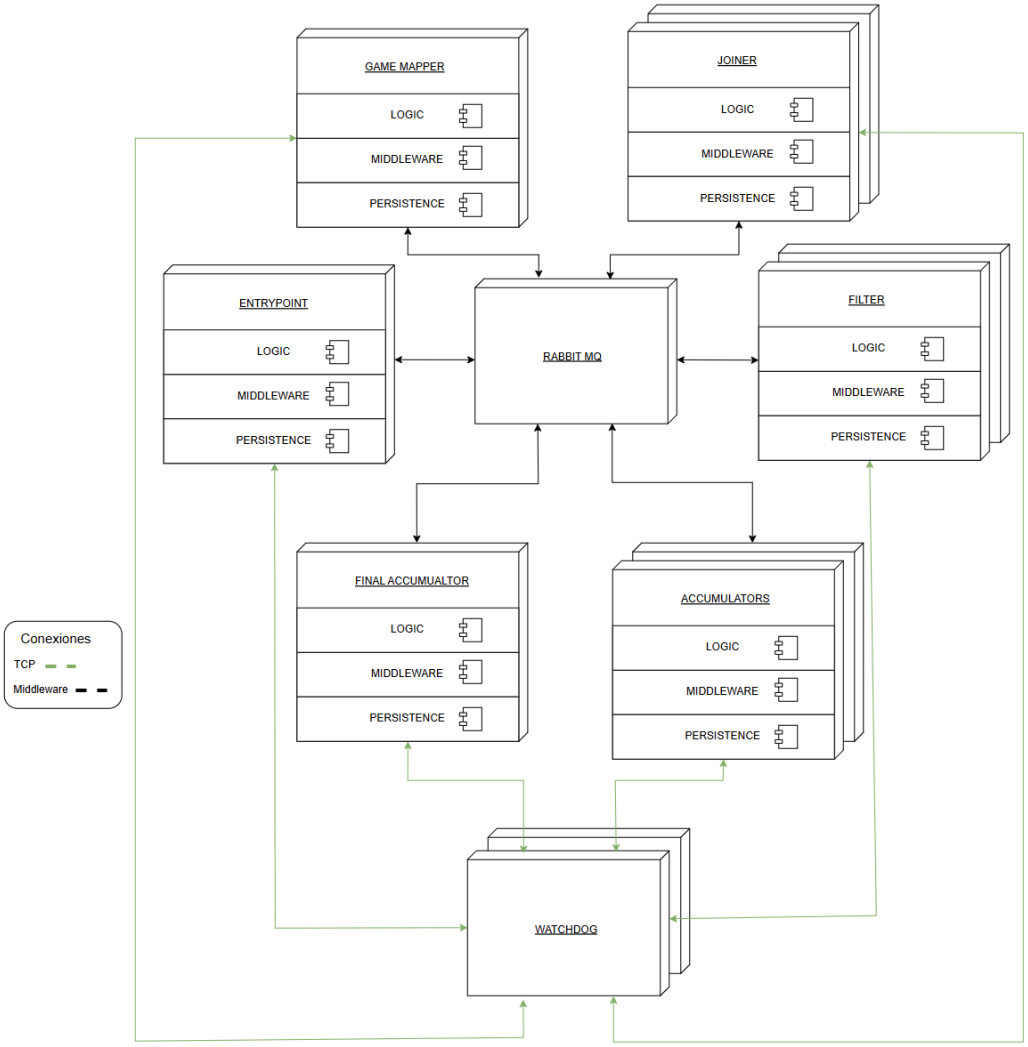


En esta versión se puede observar que algunos nodos fueron eliminados o modificados y otros re-ordenados. La Query 4 fue la consulta que más cambios sufrió: Decidimos unificar ambos joiners que teníamos casi al final de las últimas 2 consultas. Ahora se hacen al inicio. Esta decisión fue tomada a partir de la velocidad de procesamiento del English Filter por lo que en vez de hacerlo al comienzo, que significaba una ralentización en el procesamiento de toda la consulta decidimos realizar el Joiner al comienzo. Lo que provocó una optimización en cuanto a la velocidad de procesamiento de la consulta 4 completa, el english filter ahora no tiene que procesar todas las reseñas (la cantidad depende del dataset usado y en el peor de los casos, con el dataset completo, son 6 millones de consultas) sino que ahora realiza la detección de idioma únicamente para unas cuantas reseñas (las correspondientes a los juegos de acción y negativas). Actualmente por esta modificación, la consulta 4 finaliza en un tiempo adecuado, a diferencia de en la primera versión en donde hacía mucho trabajo innecesario que la retrasaba de gran manera.

Diagrama de Despliegue

En este diagrama se presenta la distribución física y lógica de los distintos componentes del sistema en una arquitectura distribuida. Los módulos que procesan datos relacionados con juegos y reseñas están distribuidos en distintos nodos que interactúan entre sí a través de colas de mensajería proporcionadas por RabbitMQ.

Cada uno de los componentes principales como el Game Mapper, los Joiners, los Accumulators y los Filtros están desplegados en nodos independientes los cuales pueden ser contenedores Docker o máquinas virtuales separadas. Cabe aclarar que en el diagrama se agruparon los nodos por funcionalidad para un entendimiento más claro del despliegue. Obviamente hay diferentes tipos de filtros o joiners que procesan diferentes mensajes pero en la base cumplen las mismas funcionalidades.



En la imagen se observa el diagrama de despliegue simplificado donde vemos que, cuando el sistema está en ejecución, los agentes del sistema se comunican entre sí a través del middleware. También

todos se conectan al agente llamado Watchdog que tiene la responsabilidad de hacer un health check a cada uno de los nodos y levantarlos si es necesario.

Tipos de nodos

En el sistema se pueden distinguir dos estrategias de procesamiento dependiendo de la necesidad del nodo.

Acumuladores

Los acumuladores reciben mensajes que deben almacenar para realizar un procesamiento sobre la totalidad de la información. Una vez que reciben y procesan toda la información se continúa enviando el o los mensajes a los nodos subsiguientes y los recursos asociados son liberados.

Filtros

Cada vez que un mensaje llega a un nodo filtro el mismo lo procesa y lo reenvía en el mismo ciclo de ejecución.

Joiners

Estos nodos acumulan el estado de los mensajes que reciben hasta encontrar un par correspondiente para realizar el join. Una vez que se encuentra el par, los mensajes se procesan y el resultado es reenviado al nodo subsiguiente. Después de completar esta operación, el espacio ocupado por los mensajes procesados es liberado, optimizando el uso de recursos.

Separación en capas

A nivel de estructura de código, en los nodos pertenecientes al sistema, se implementó una separación en capas. En cada nodo contamos con tres capas: ***lógica***, ***middleware*** y ***persistencia***.

Lógica

Esta capa contiene la implementación principal de las reglas de negocio y el procesamiento específico que realiza el nodo. Aquí se define cómo se transforman y manipulan los datos según los requisitos del sistema.

Middleware

Actúa como un intermediario entre la lógica y los mecanismos de comunicación. Gestiona la recepción y envío de mensajes y las interacciones con colas y exchanges del middleware.

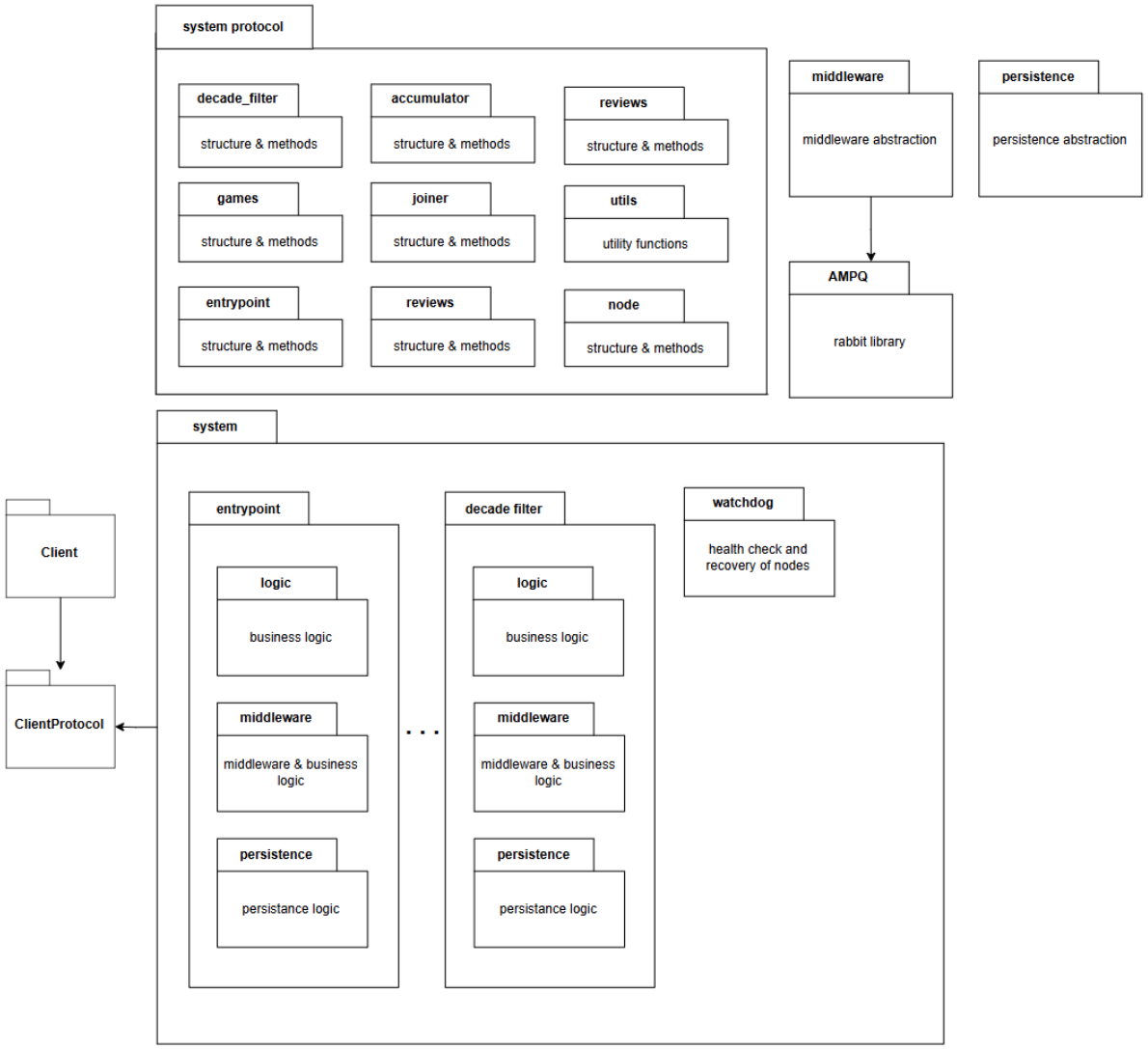
Persistencia

Se encarga de almacenar los datos de manera permanente o temporal. Esto incluye estructuras que contengan información del estado de la query y del mecanismo de registro de mensajes que implementa el sistema permitiendo que los nodos guarden su estado entre operaciones y recuperen información si el mismo se recupera de una caída.

Vista De Desarrollo

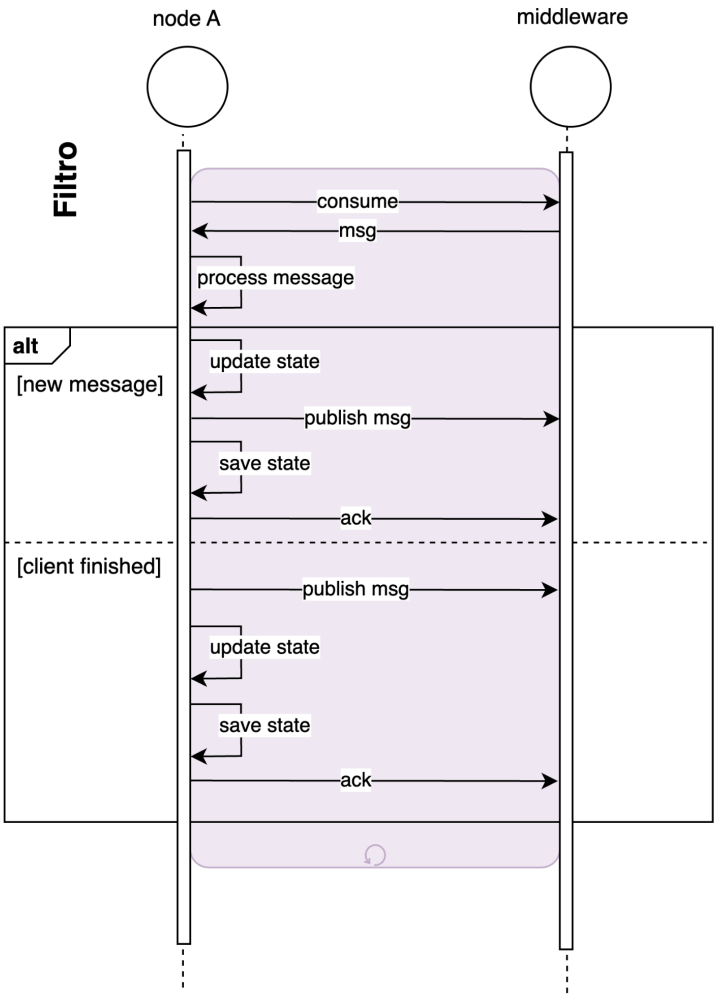
Diagrama de Paquetes

Esta separación facilita la mantenibilidad del código, permite una mayor flexibilidad en la adaptación a cambios y asegura una clara división de responsabilidades en cada nodo del sistema.



Middleware

La comunicación entre los nodos se realiza mediante un middleware que proporciona abstracciones como exchanges y colas facilitando el flujo de mensajes a través del sistema. En esta implementación se utilizó el framework RabbitMQ. Los nodos de nuestro sistema publican sus mensajes a exchanges definidos por el mismo y consumen los mensajes a través de colas también definidas por el mismo sistema.



El diagrama de secuencia ilustra la interacción de un nodo de tipo filtro con el middleware. Se pueden apreciar las acciones que se desencadenan en el nodo a partir de que el mismo consume un mensaje del middleware. Los mensajes *update state* y *save state* indican que se está actualizando o guardando el estado actual del nodo, esto siempre se hace antes de enviar el *ack* del mensaje.

Abstracción del Middleware

El middleware desempeña un papel central en la arquitectura de nuestro sistema, actuando como la capa de comunicación que permite la interacción entre los nodos distribuidos. Se implementó un MiddlewareManager que encapsula las operaciones necesarias para gestionar exchanges, colas y sus interacciones en RabbitMQ. A continuación, detallamos los aspectos más relevantes de esta abstracción:

Diseño del Middleware Manager

El componente principal, MiddlewareManager, simplifica la gestión de la comunicación al abstraer las operaciones con RabbitMQ. El constructor NewMiddlewareManager establece la conexión inicial con RabbitMQ, asegurando que el middleware esté accesible durante toda la ejecución del nodo.

Creación y Configuración de Colas

El MiddlewareManager facilita la creación y configuración de colas en RabbitMQ mediante métodos dedicados que garantizan una inicialización consistente y modular. Cada cola puede configurarse con atributos personalizados como nombres, claves de enrutamiento. Por ejemplo, al crear una cola vinculada a un exchange, el método CreateBoundQueue permite asociarla con una clave de enrutamiento específica. Este mecanismo asegura que los mensajes publicados en el exchange con dicha clave sean entregados a la cola correspondiente.

Gestión de Exchanges

El MiddlewareManager abstrae la configuración de exchanges, proporcionando métodos como CreateExchange para definir diferentes tipos de intercambio (directo, fanout, topic, etc.). Esta abstracción asegura que los mensajes sean distribuidos de acuerdo con los patrones de enrutamiento definidos.

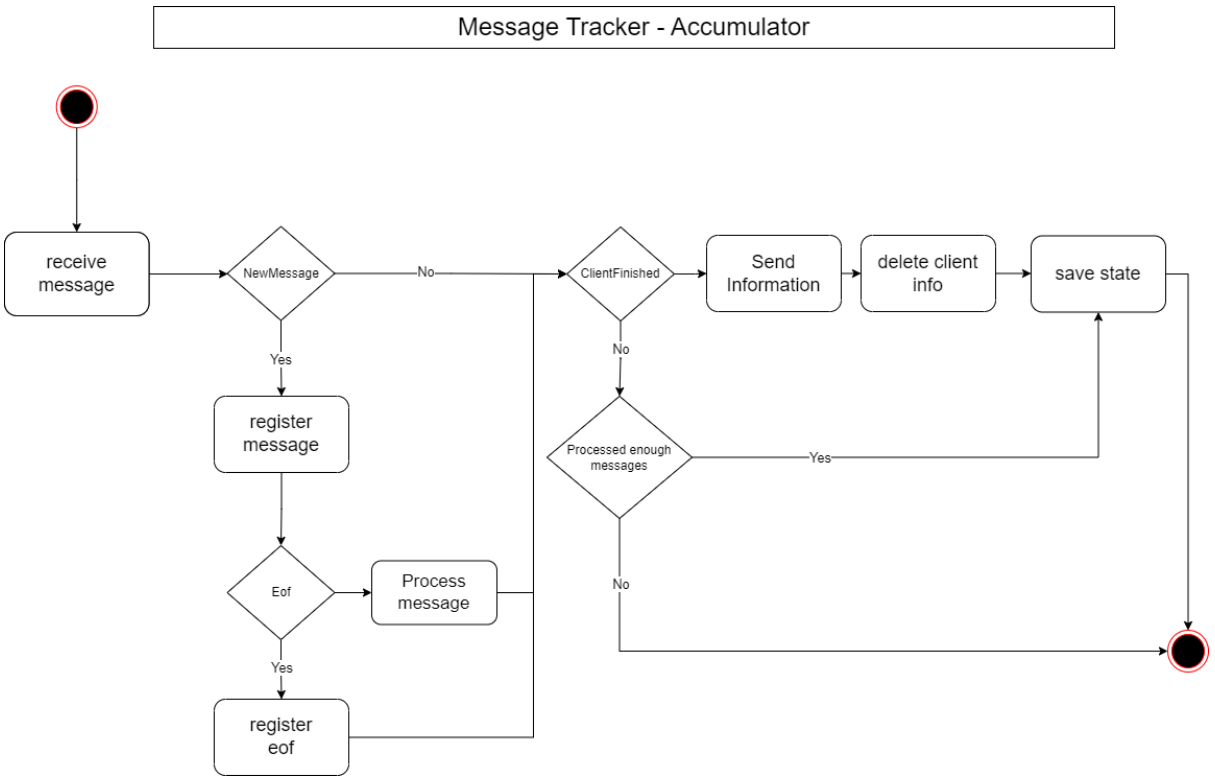
Vista de Procesos

Registro de mensajes entre nodos

Se implementó una estructura llamada Message Tracker cuyo objetivo es garantizar el seguimiento y control del estado de los mensajes procesados en un nodo determinado para cada cliente. El **Message Tracker** se implementó como una estructura de datos que permite registrar los mensajes procesados, los mensajes enviados, y los indicadores de finalización (EOFs) asociados a cada cliente. Con el fin de garantizar la persistencia del estado del Message Tracker, se implementaron funciones para su serialización y deserialización. Esto permite transformar su contenido en un formato binario y almacenarlo manteniendo la coherencia de los datos ante caídas.

Cada mensaje envíado por el nodo es registrado en esta estructura distinguiendo a qué cliente corresponde. Una vez que fueron envíados todos los mensajes el cliente manda un mensaje de tipo EOF que contiene la información de la cantidad de mensajes que debería esperar al siguiente nodo. Asimismo el nodo que recibe este mensaje procesa los mensajes entrantes distinguiendo también por cliente y registrando qué mensajes fueron recibidos. En este proceso se aplica una función de hash a cada mensaje con el objetivo de identificar si el mensaje fue procesado anteriormente o no.

Un nodo considera que ha terminado con un cliente cuando la cantidad de mensajes esperados, que fue recibido dentro del EOF del nodo anterior, es igual a la cantidad de mensajes procesados por el mismo. De esta manera aseguramos la trazabilidad de los mensajes que se envían entre los nodos.

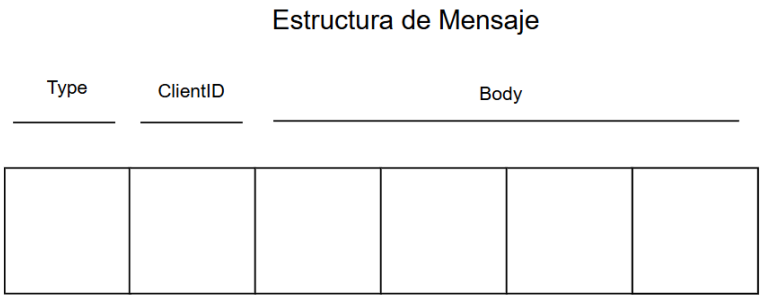


En la imagen podemos observar un diagrama de actividades para la estructura message tracker correspondiente a un nodo de tipo acumulador. El mismo almacena los mensajes hasta que la condición de Client Finished se cumple. Una vez cumplida, se toma la información de la cantidad de mensajes enviados para armar el EOF que recibirá el siguiente nodo. Finalmente se elimina la información del cliente y se guarda el estado de la estructura.

Tipos de mensajes

El sistema posee diversos tipos de mensajes para poder adaptarse a los requerimientos específicos de cada nodo pero en este caso estaremos hablando de la conformación de los diferentes tipos de mensajes y cómo se construyen.

La estructura básica de un mensaje puede representarse de esta forma:



La acción de deserializar un mensaje se hará en 3 partes: Tipo, ID de cliente y el cuerpo o body del mensaje.
Según el tipo de mensaje será la forma del procesamiento del mensaje recibido. Además de los distintos mensajes con estructuras del negocio que varían de nodo en nodo, los siguientes son mensajes que se manejan en la mayoría de los nodos del sistema:

End Of File

Refleja cuando el nodo previo al actual termina de procesar todos los mensajes de un cliente por lo que es la forma que los nodos tienen para notificar que pueden continuar con el procesamiento. Además los mensajes de este tipo gracias al MessageTracker explicado anteriormente contienen en su cuerpo los mensajes enviados para asegurar que efectivamente todos los mensajes enviados hayan sido procesados por el siguiente nodo.

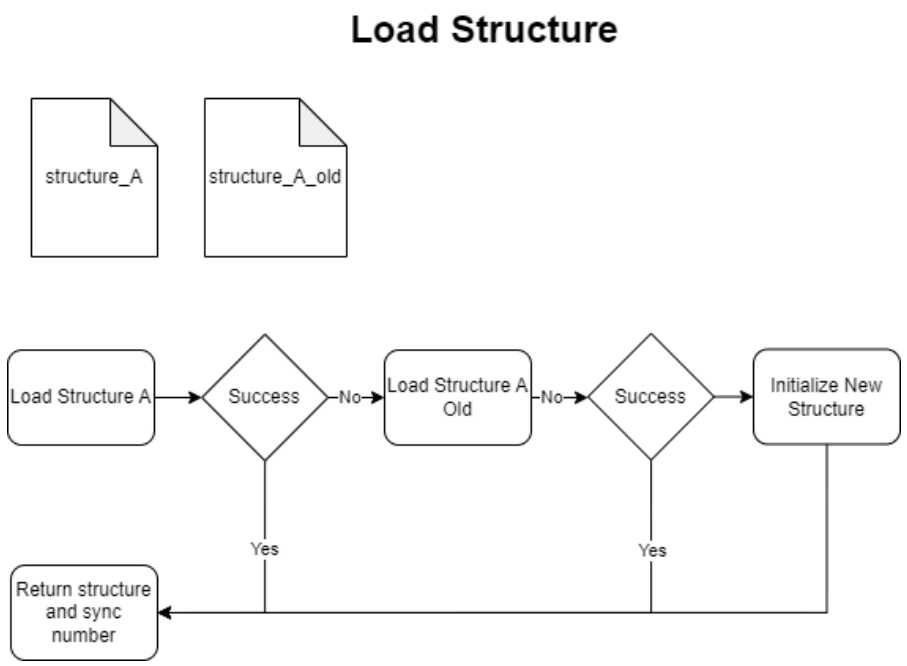
Delete Client

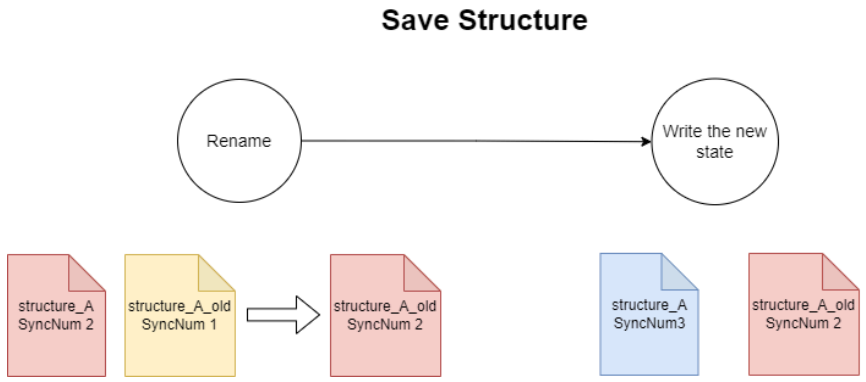
Cuando el Entrypoint sufre una falla y cae entonces todos los clientes que no hayan finalizado deberán ser limpiados del sistema junto con todos sus mensajes por lo que un nodo al recibir un mensaje de este tipo procederá a eliminar todos los mensajes asociados al cliente recibido y propagar el mensaje a los siguientes nodos.

Persistencia y Consistencia

La implementación del módulo de persistencia está diseñada con un enfoque genérico y extensible que permite manejar cualquier estructura de datos que cuente con funciones de serialización y deserialización. Esto hace que el sistema sea flexible y capaz de adaptarse a nuevas necesidades sin requerir modificaciones significativas en su infraestructura.

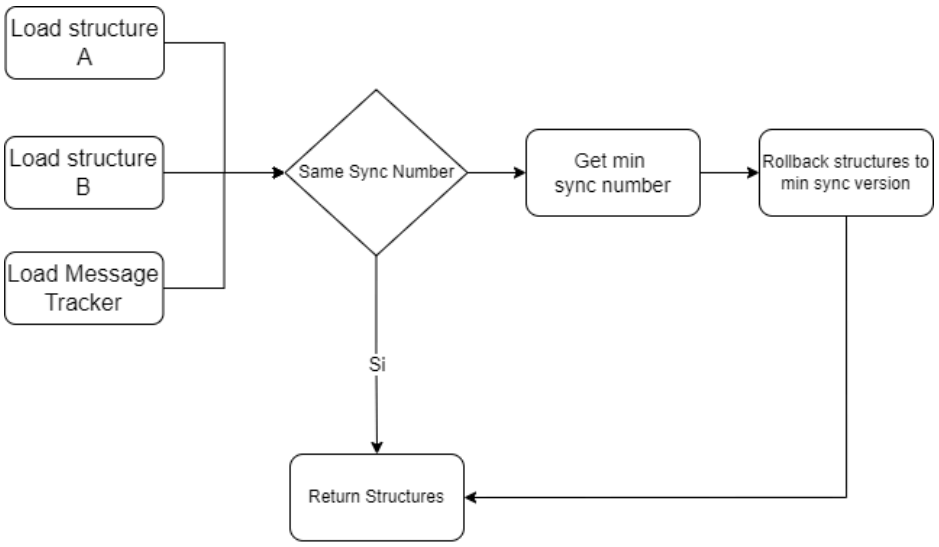
La persistencia se logra mediante la estructura genérica Persister, que abstrae las operaciones de almacenamiento y recuperación. Esta estructura permite guardar los datos en un formato binario estructurado que incluye un número de sincronización (syncNumber), la longitud de los datos serializados y los datos en sí. Antes de sobrescribir el archivo principal, se crea un respaldo (_old), lo que asegura que los datos previos estén disponibles para recuperación en caso de fallos. Este enfoque no solo protege la integridad de los datos, sino que también facilita la implementación de mecanismos de recuperación ante errores.





El sistema de persistencia incluye mecanismos para mantener la consistencia de los datos entre estructuras relacionadas. Cuando varias estructuras se guardan conjuntamente, se sincronizan utilizando el menor número de sincronización compartido entre ellas. Si se detecta una desalineación entre los números de sincronización, el sistema recurre a los archivos de respaldo para restaurar un estado coherente. Esto garantiza que todas las estructuras involucradas operen de manera alineada y confiable, incluso en escenarios donde ocurran interrupciones inesperadas.

Otro aspecto importante es la capacidad de realizar operaciones de recuperación dual. Al cargar datos, el sistema primero intenta recuperar los datos del archivo primario. Si esta operación falla, se recurre automáticamente al respaldo. Además, ante inconsistencias detectadas, se aplica un rollback automático desde los archivos de respaldo, lo que permite restaurar el sistema a un estado coherente.



La arquitectura también incluye operaciones genéricas como SaveAll y LoadAll, que permiten guardar y recuperar múltiples estructuras de manera conjunta, garantizando que todas permanezcan sincronizadas bajo un mismo número de sincronización. Esto simplifica la gestión de estructuras relacionadas y previene inconsistencias.

El diseño de la tolerancia a fallas se centra en la operación crucial del guardado de datos. Con la ayuda de la estructura del message tracker que fue explicado anteriormente logramos que el trabajo fuera tolerante a fallas. Con esto, los datos siempre están actualizados y sincronizados entre estructuras. Al sufrir una caída, aquellos mensajes que no hayan sido guardados serán recuperados/ re-encolados por ser mensajes que no hayan podido ser “ackeados”. Aquellos mensajes repetidos que hayan sido procesados exitosamente no serán procesados nuevamente gracias al Message Tracker que como bien su nombre explícita, entre varias tareas es el encargado de llevar un registro de aquellos mensajes que efectivamente hayan sido procesados y guardados exitosamente.

Watchdog

Para detectar servicios caídos, el programa implementa un algoritmo de elección de líder conocido como *Bully*. Este mecanismo permite que un grupo de nodos distribuyan roles de liderazgo de forma dinámica, garantizando la continuidad del servicio en caso de que falle el nodo coordinador.

El nodo coordinador es responsable de monitorear la disponibilidad de los demás nodos mediante pings periódicos. Si detecta que un nodo no responde, lo reinicia utilizando *Docker in Docker*. En caso de que el coordinador falle, los nodos restantes inician automáticamente un proceso de elección para designar a un nuevo líder.

Cada nodo cuenta con un ID numérico estático, y el nodo disponible con el ID más alto es siempre elegido como coordinador, asegurando un proceso de selección predecible.

Conclusión

A lo largo del desarrollo del trabajo práctico logramos implementar un sistema distribuido capaz de procesar grandes volúmenes de datos de reseñas de videojuegos en Steam, cumpliendo con los objetivos de tolerancia a fallos y escalabilidad. El enfoque modular basado en nodos con roles específicos, la implementación de un middleware robusto (RabbitMQ) y el diseño de mecanismos de persistencia y consistencia garantizaron un sistema confiable y alineado con los principios teóricos del curso.

Entre los principales logros, destacamos la correcta separación en capas, que permitió una mayor claridad y mantenibilidad del código, y el diseño de estructuras como el Message Tracker y el Persister, que aseguraron la trazabilidad de los mensajes y la consistencia de los datos en escenarios de fallo. Asimismo, las iteraciones del proyecto facilitaron mejoras significativas en el diseño del diagrama de robustez y la optimización de las consultas planteadas.