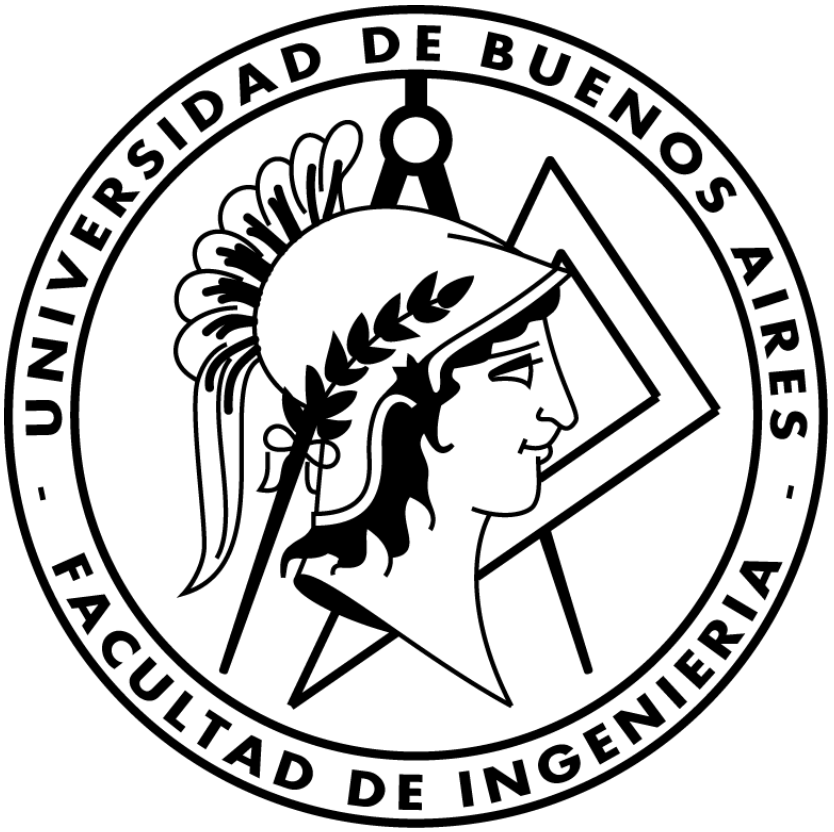


Trabajo Práctico Grupal

Steam Analysis

Sistemas Distribuidos I (75.74)

Fecha de Entrega: **Jueves 31/10/2024**

Francisco O. Lorda	105554	forqueral@fi.uba.ar
Carolina Di Matteo	103963	cdimatteo@fi.uba.ar
Tomás Apaldetti	105157	tapaldetti@fi.uba.ar

Índice

Scope del Proyecto

Este proyecto está orientado al diseño y desarrollo de un sistema distribuido que permita procesar y analizar las reseñas de juegos de la plataforma Steam. El sistema tiene como objetivo gestionar grandes volúmenes de datos, incluyendo información sobre los juegos, como plataformas compatibles, reseñas de usuarios, géneros, y diversas métricas relacionadas con el tiempo promedio de juego, el número de reseñas positivas y negativas, entre otras. El sistema está diseñado para generar resultados analíticos sobre la cantidad de juegos disponibles en diferentes plataformas, así como para listar juegos que cumplan con ciertos criterios, tales como género o métricas específicas.

Entre los principales objetivos funcionales se destacan:

- Procesar y clasificar los juegos según la plataforma en la que están disponibles
- Obtener listas de juegos de los géneros “Indie” y “Shooter”, basadas en diversas métricas predefinidas

Arquitectura del Software

El sistema se basa en un modelo distribuido Cliente-Servidor, organizado de la siguiente manera:

Clientes

Su principal responsabilidad es enviar datasets al servidor en formato binario. Estos clientes pueden ejecutarse de manera distribuida, permitiendo múltiples instancias, pero no realizan ningún tipo de procesamiento local de los datos.

Servidor

Es el componente encargado de recibir los datasets desde los clientes. El servidor almacena temporalmente los datos necesarios para procesar las consultas y utiliza procesamiento en paralelo (mediante multithreading) para gestionar múltiples solicitudes simultáneamente. Además, el servidor debe ser capaz de procesar los datos de manera incremental, lo que significa que debe comenzar a procesar cada dataset tan pronto como los reciba, sin necesidad de esperar a recibir la totalidad de los datos.

Middleware

Actúa como intermediario entre los clientes y el servidor, facilitando la transferencia de datasets. Este componente implementará un sistema de mensajería, como RabbitMQ, que permitirá la distribución y el encolado eficiente de las consultas y datasets entre los clientes y el servidor, garantizando una comunicación efectiva y robusta.

Metas

Procesamiento Incremental

El Servidor debe ser capaz de procesar datos a medida que los recibe, sin requerir que se almacene o cargue todo el dataset antes de iniciar el procesamiento.

Escalabilidad

La arquitectura del sistema debe ser escalable, permitiendo la adición de más clientes y servidores a medida que aumente el volumen de datos y consultas a procesar.

Distribución Eficiente de la Carga

El Middleware debe asegurar que los datasets y las consultas se distribuyan eficientemente entre los diferentes servidores, evitando así la sobrecarga en un solo nodo.

Manejo de SIGTERM

El Sistema debe implementar un manejo adecuado de señales SIGTERM, permitiendo que el servidor finalice de manera ordenada el procesamiento de consultas antes de cerrar.

Portabilidad y Flexibilidad

El Sistema debe garantizar portabilidad y flexibilidad en diferentes entornos de ejecución, asegurando que pueda ser desplegado de manera eficiente y agnóstica a la infraestructura subyacente, ya sea en términos de procesadores físicos o virtuales. Adicionalmente, debe mantener compatibilidad con sistemas Linux y utilizar tecnologías como Docker para facilitar un despliegue consistente y reproducible, independientemente de la plataforma o configuración.

Limitaciones

- El Servidor no almacenará el contenido de cada uno de los datasets enviados por los Cliente, sino únicamente la información necesaria para el procesamiento de las consultas.
- Cualquier nodo trabajador no puede asumir que puede cargar el dataset en memoria.
- El Servidor no conoce el contenido de los datasets, pero sí sus correspondientes estructuras.
- La capacidad de cómputo de cada nodo es limitada (tanto procesamiento como almacenamiento en memoria).
- El Sistema responde únicamente a las queries enunciadas.
- El Sistema utiliza Message Oriented Middleware como método de comunicación entre nodos; en particular: RabbitMQ.
- El Sistema se desarrollará en Golang.

Escenarios

Casos de Uso

Para enunciar los casos de uso, utilizaremos el siguiente diccionario de abreviaturas:

- ➢ Web Server = WS
- ➢ Worker Map & Filter de Juegos = WG_{MF}
- ➢ Worker Map & Filter de Reseñas = WR_{MF}
- ➢ Worker Query i = WQ_i, i = 1...5

Obtener Total de Juegos por Plataforma

Descripción: El Sistema debe devolver la cantidad de juegos soportados en cada plataforma (Windows, Linux, MAC).

Actores: Cliente, WS, WG_{MF}, WQ_1.

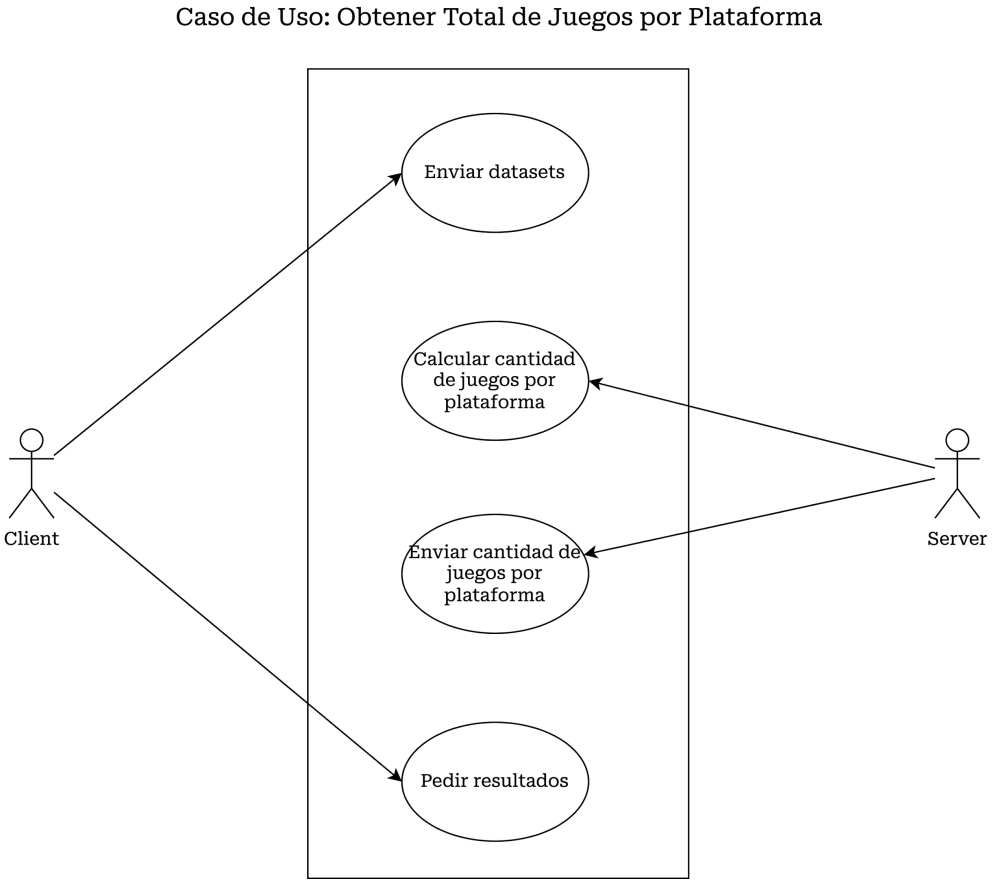
Precondiciones: El Cliente envía el dataset de juegos.

Flujo normal:

- El Cliente envía el dataset de juegos al WS.
- El WS envía el dataset al WG_{MF}.
- El WG_{MF} envía el dataset al WQ_1.
- El WQ_1 calcula la cantidad de juegos por plataforma.
- El WQ_1 almacena los resultados en su temporal storage y los reenvía al WS.
- El WS recibe los resultados y los envía al Cliente.

Flujo alternativo: Si hay problemas de comunicación con un Worker, se intenta reenviar la solicitud.

Postcondiciones: El Cliente recibe la cantidad de juegos por plataforma.



Obtener Top 10 Juegos más jugados Género “Indie” de 2010

Descripción: El Sistema debe devolver los 10 juegos del género “Indie” con más tiempo promedio de juego publicados en la década de 2010.

Actores: Cliente, WS, WG_{MF}, WQ_2.

Precondiciones: El Cliente envía el dataset de juegos.

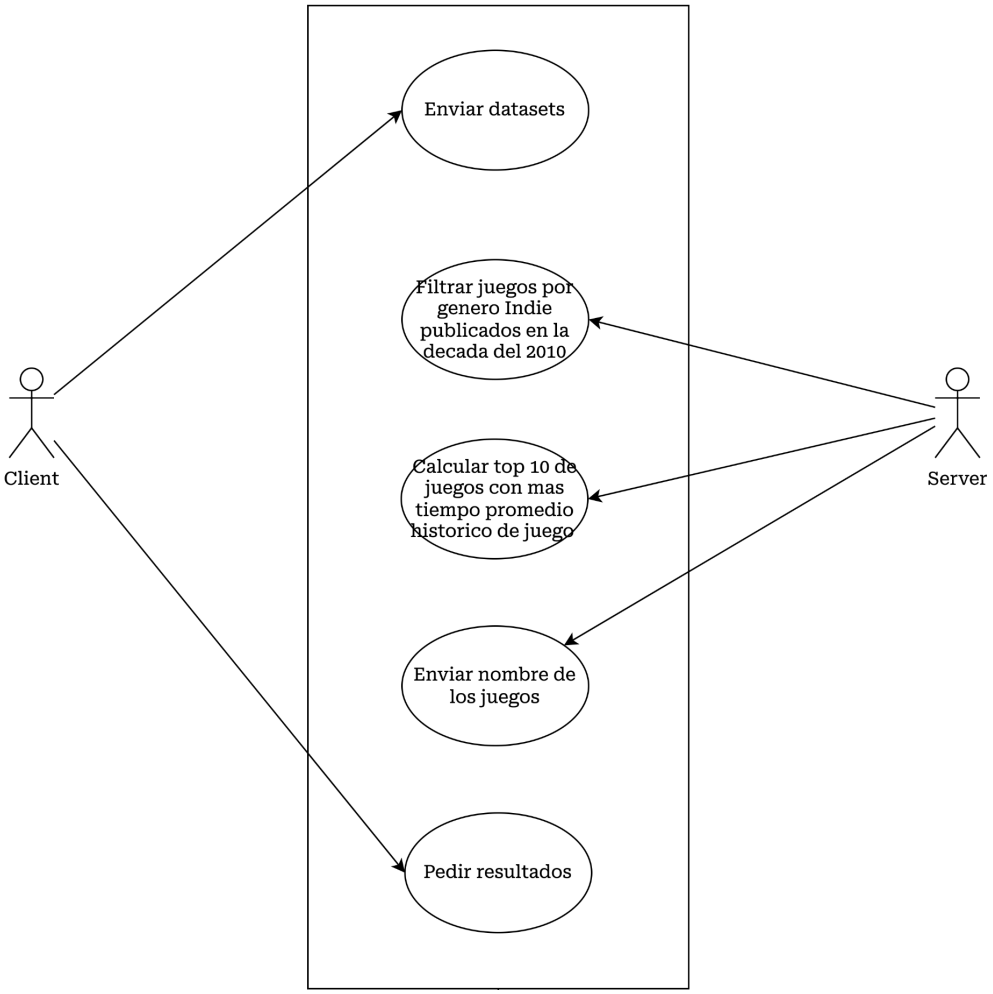
Flujo normal:

- El Cliente envía el dataset de juegos al WS.
- El WS envía el dataset al WG_{MF}.
- El WG_{MF} filtra los juegos del género “Indie” publicados en 2010 y los envía al WQ_2.
- El WQ_2 selecciona los 10 juegos con mayor tiempo promedio de juego.
- El WQ_2 almacena los resultados en su temporal storage y los reenvía al WS.
- El WS recibe los resultados y los envía al Cliente.

Flujo alternativo: Si no hay suficientes datos, se devuelve un mensaje al cliente.

Postcondiciones: El Cliente recibe la lista de los 10 juegos “Indie”.

Caso de Uso: Obtener Top 10 Juegos más jugados “Indie” de 2010



Obtener Top 5 Juegos Género “Indie” con más Reseñas Positivas

Descripción: El Sistema debe devolver los nombres de los 5 juegos del género “Indie” con más reseñas positivas.

Actores: Cliente, WS, WG_{MF}, WR_{MF}, WQ_3.

Precondiciones: El Cliente envía los datasets de juegos y reviews.

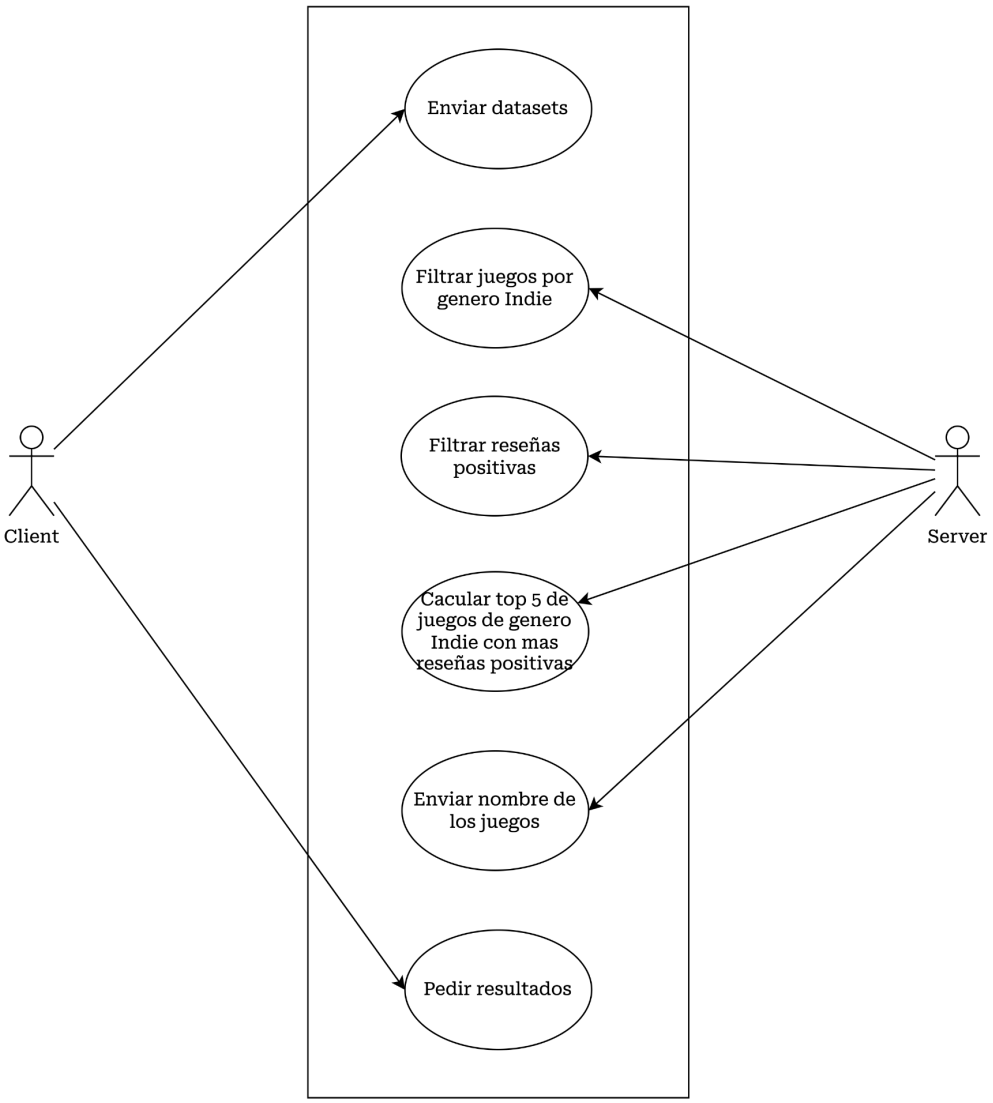
Flujo normal:

- El Cliente envía los datasets de juegos al WS.
- El WS envía el dataset al WG_{MF}.
- El WS envía el dataset al WR_{MF}.
- El WR_{MF} filtra las reseñas positivas y los envía al WQ_3.
- El WG_{MF} filtra los juegos del género “Indie” y los envía al WQ_3.
- El WQ_3 selecciona los top 5 juegos de género “Indie” con más reseñas positivas.
- El WQ_3 almacena los resultados en su temporal storage y los reenvía al WS.
- El WS recibe los resultados y los envía al Cliente.

Flujo alternativo: Si no hay suficiente información de reseñas, se devuelven menos de 5 resultados.

Postcondiciones: El Cliente recibe el Top de los 5 juegos de género “Indie” con más reseñas positivas.

Caso de Uso: Obtener Top 5 Juegos “Indie” con más Reseñas Positivas



Obtener Juegos “Action” con más de 5,000 Reseñas Negativas en Inglés

Descripción: El Sistema debe devolver los juegos del género “Action” con más de 5,000 reseñas positivas en inglés.

Actores: Cliente, WS, WG_{MF}, WR_{MF}, WQ_4.

Precondiciones: El Cliente envía los datasets de juegos y reviews.

Flujo normal:

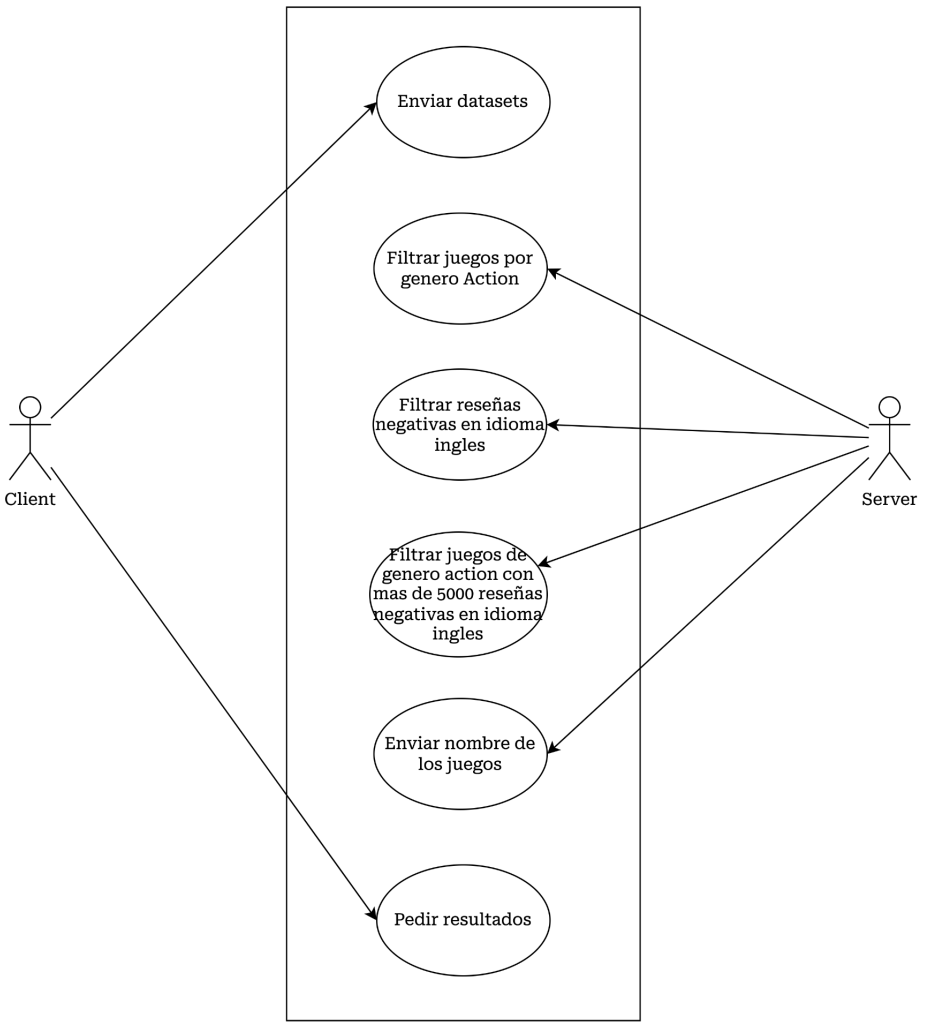
- El cliente envía los datasets de juegos al WS.
- El WS envía el dataset al WG_{MF}.
- El WS envía el dataset al WR_{MF}.
- El WR_{MF} filtra las reseñas negativas en idioma inglés y las envía al WQ_4.

- El WG_{MF} filtra los juegos del género “Action” y los envía al WQ_4.
- El WQ_4 selecciona los juegos shooter con más de 5,000 reseñas positivas en inglés
- El WQ_4 almacena los resultados en su temporal storage y los reenvía al WS.
- El WS recibe los resultados y los envía al Cliente.

Flujo alternativo: Si hay menos de 5,000 reseñas, se devuelve un error o una lista vacía.

Postcondiciones: El Cliente recibe la lista de juegos “Action” con más de 5,000 reseñas negativas en inglés.

Caso de Uso: Obtener Juegos “Action” con más de 5,000 Reseñas Negativas en Inglés



Obtener Juegos “Action” en el Percentil 90 de Reseñas Negativas

Descripción: El Sistema debe devolver los nombres de los juegos del género “Action” con mayor cantidad de reseñas negativas, en el percentil 90.

Actores: Cliente, WS, WG_{MF}, WR_{MF}, WQ_5.

Precondiciones: El Cliente envía los datasets de juegos y reseñas.

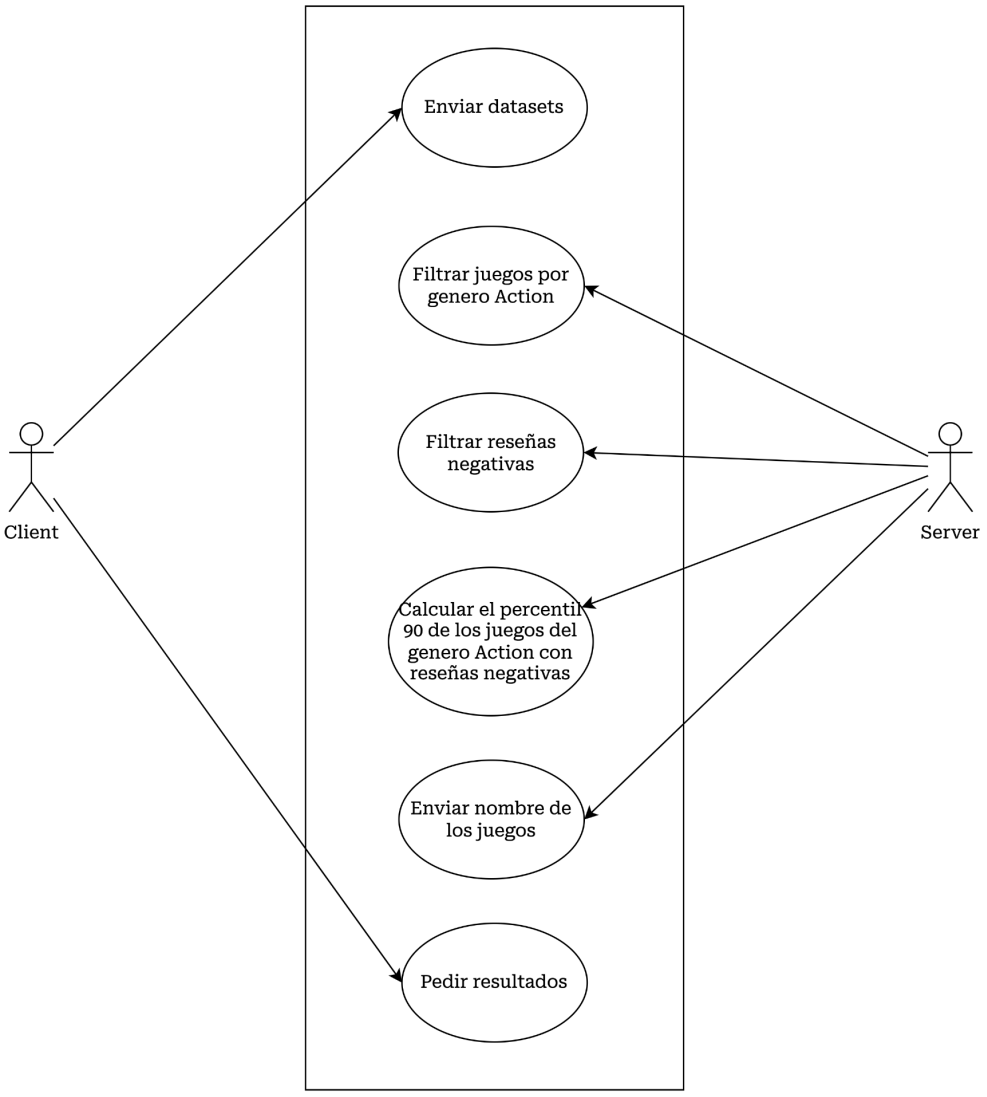
Flujo normal:

- El Cliente envía los datasets al WS.
- El WS envía los juegos al WJ_{MF} y las reseñas al WR_{MF}.
- El WR_{MF} filtra las reseñas negativas y las envía al WQ_5.
- El WG_{MF} filtra los juegos del género “Action” y los envía al WQ_5.
- El WQ_5 procesa los datos para calcular el percentil 90 de reseñas negativas.
- El WQ_5 almacena los resultados en su temporal storage y los reenvía al WS.
- El WS recibe los resultados y los envía al Cliente.

Flujo alternativo: Si no hay suficientes reseñas, se devuelve un mensaje de error.

Postcondiciones: El Cliente recibe la lista de juegos “Action” en el Percentil 90 de Reseñas Negativas.

Caso de Uso: Obtener Juegos “Action” en el Percentil 90 de Reseñas Negativas



Protocolo Cliente-Servidor

En primer lugar, decidimos que el Cliente y el Servidor se comunicarán vía TCP. Además, para establecer una comunicación confiable y evitar los fenómenos *short read* y *short write* decidimos adicionar una cabecera de 4 bytes con la información del largo del contenido del mensaje efectivamente.

De esta forma, el emisor del mensaje tiene la tarea de calcular el largo del mensaje a enviar, y escribir dicha información en el *buffer*; luego, se agrega efectivamente el contenido final.

El receptor, leerá los primeros 4 bytes del mensaje entrante, lo que le indicará la longitud de bytes que tiene que leer del buffer para obtener la información que el emisor quiere transmitir.

Ahora bien, para implementar esta funcionalidad utilizamos la siguiente estructura de mensaje, la cual permite ser Serializada -por quien escribirá un mensaje por el socket TCP- y Deserializada -por quien leerá el mensaje que arriba al socket TCP-:

Inicialmente, el Cliente se conecta al socket del Servidor, quien estará esperando por nuevas conexiones. El servidor generará un código único (*uuid*) que asociará a dicho Cliente, y con el cual lo reconocerá transversalmente durante el flujo del sistema.

Una vez inicializada la configuración del Cliente desde el Servidor, este le enviará el código generado. El Cliente almacenará esta información de forma tal que en el futuro le permita reestablecer su sesión con el Servidor en caso de alguna falla, o bien consultar por los resultados del análisis.

Luego, el Cliente comenzará a enviar el contenido de los *datasets* de Juegos y Reseñas de forma paralela. Para esto, serializará el mensaje con el tipo y el contenido de cada uno según corresponda. Esto se visualizará como: `REV|content` para reseñas, ó, `GAM|content` para juegos. El Servidor leerá estos mensajes y los deserializará para comprender de qué información se trata.

Cuando el Cliente finalice el envío de información, envía un mensaje especial con el contenido `EOF` que le permitirá al Servidor saber cuándo finaliza la transmisión de datos tanto para juegos como para reseñas. Esto se visualizará como: `REV|EOF` para reseñas, y, `GAM|EOF` para juegos.

En este punto, el Cliente solicitará los resultados de la ejecución, para lo cual enviará un mensaje de tipo `RES|uuid` incorporando al mensaje el código que oportunamente le proporcionó el Servidor tal que este pueda ubicar los resultados de forma certera.

Cuando el Servidor tenga disponible dichos resultados, comenzará a enviarlos con el formato que corresponda a cada una de las *queries*, cabe aclarar que para cada una existe un mensaje particular que le indica al Cliente a qué *query* corresponde cada resultado.

Al finalizar, el Servidor enviará al Cliente un mensaje para indicarle que finalizó el envío de resultados. Cuando el Cliente reciba todos los resultados -es decir: el mensaje que confirma la finalización del envío de resultados- enviará al Servidor un mensaje para indicar el fin de la comunicación. Esto se visualizará como: `CLC|uuid`.

Protocolo Server-Worker

El *endianness* de los mensajes es *network endianness / big endian*.

JobID: Identificador de un trabajo para un cliente en particular. Es usado para identificar un stream de datos.

_type: Identifica el tipo de Mensaje:

- Data: Se envía al handler configurado para el controller
- Control: Usado por el runtime para tareas de control. Por ahora solo existen mensajes de EOF

Content: El contenido del mensaje, esto se pasará al handler para que pueda parsear su mensaje. El protocolo del handler es independiente del protocolo del *runtime*

Vista Lógica

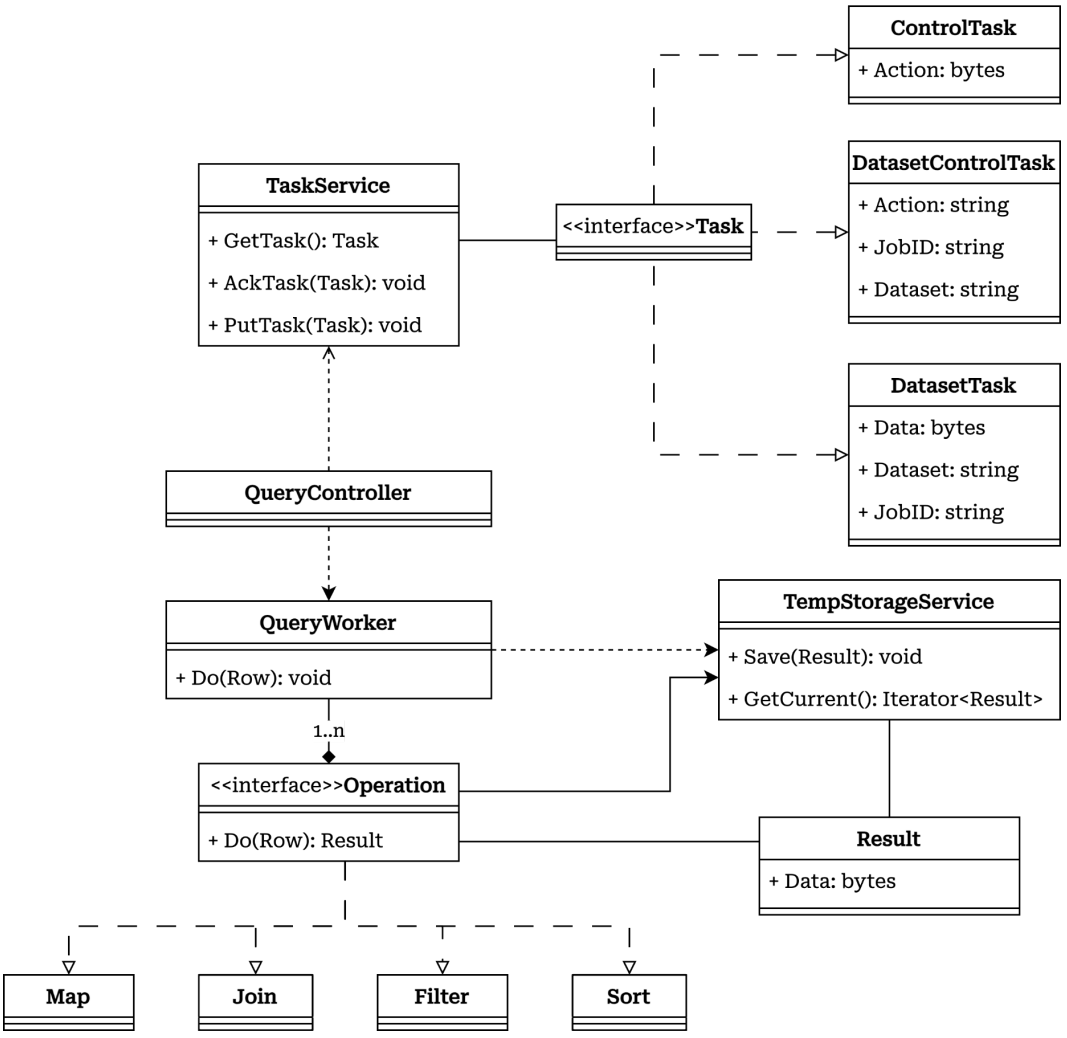
Diagramas de Clases

Worker

La clase `QueryController` será la encargada de coordinar el trabajo dentro del worker de la query en particular. Esto es:

- recibir una tarea (hacerle `unmarshall`),
- buscar la información necesaria para el procesamiento de la query,
- enviarla a la clase que realiza el trabajo, y
- realizar el `ACK` a la tarea cuando el trabajo se haya completado.

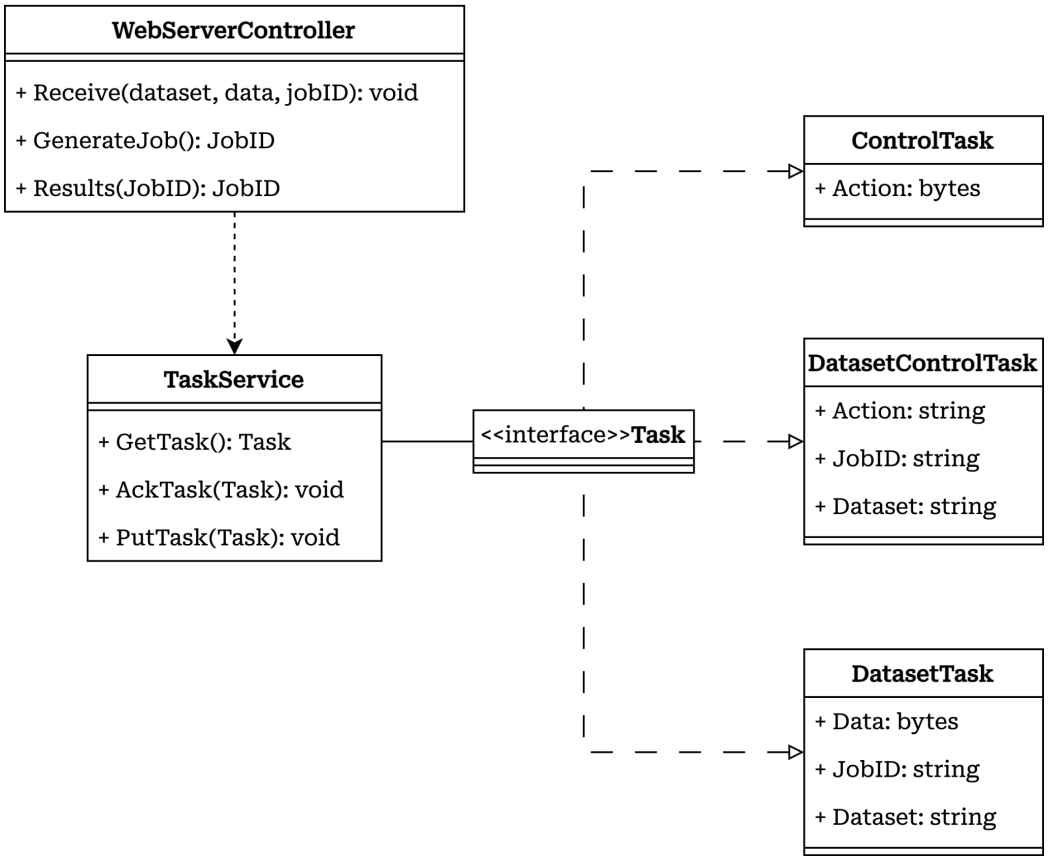
Diagrama de Clases - Worker



WebService

La clase `WebServerController` será el *boundary* entre el Cliente y el sistema en general, y se encargará de coordinar los trabajos a realizar. En esta línea, existirán tareas de control general para los diferentes *workers*.

Diagrama de Clases - WebService

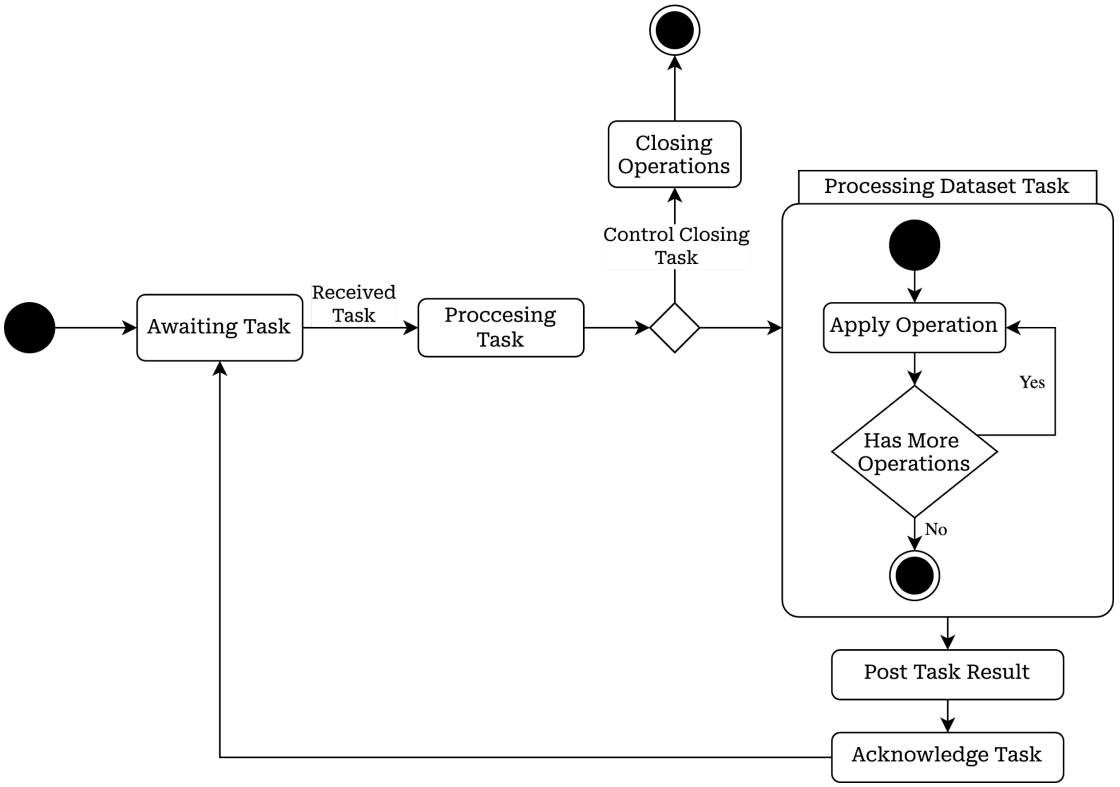


Diagramas de Estados

A continuación enunciamos únicamente el diagrama de estados de un *worker* genérico. Este componente se encontrará en un *loop* constante durante el cual se encargará de realizar tantas tareas como se le pidieran hacer.

Pueden existir tareas que rompan el flujo normal del *worker* para realizar tareas de control.

Diagrama de Estado de un Worker Genérico



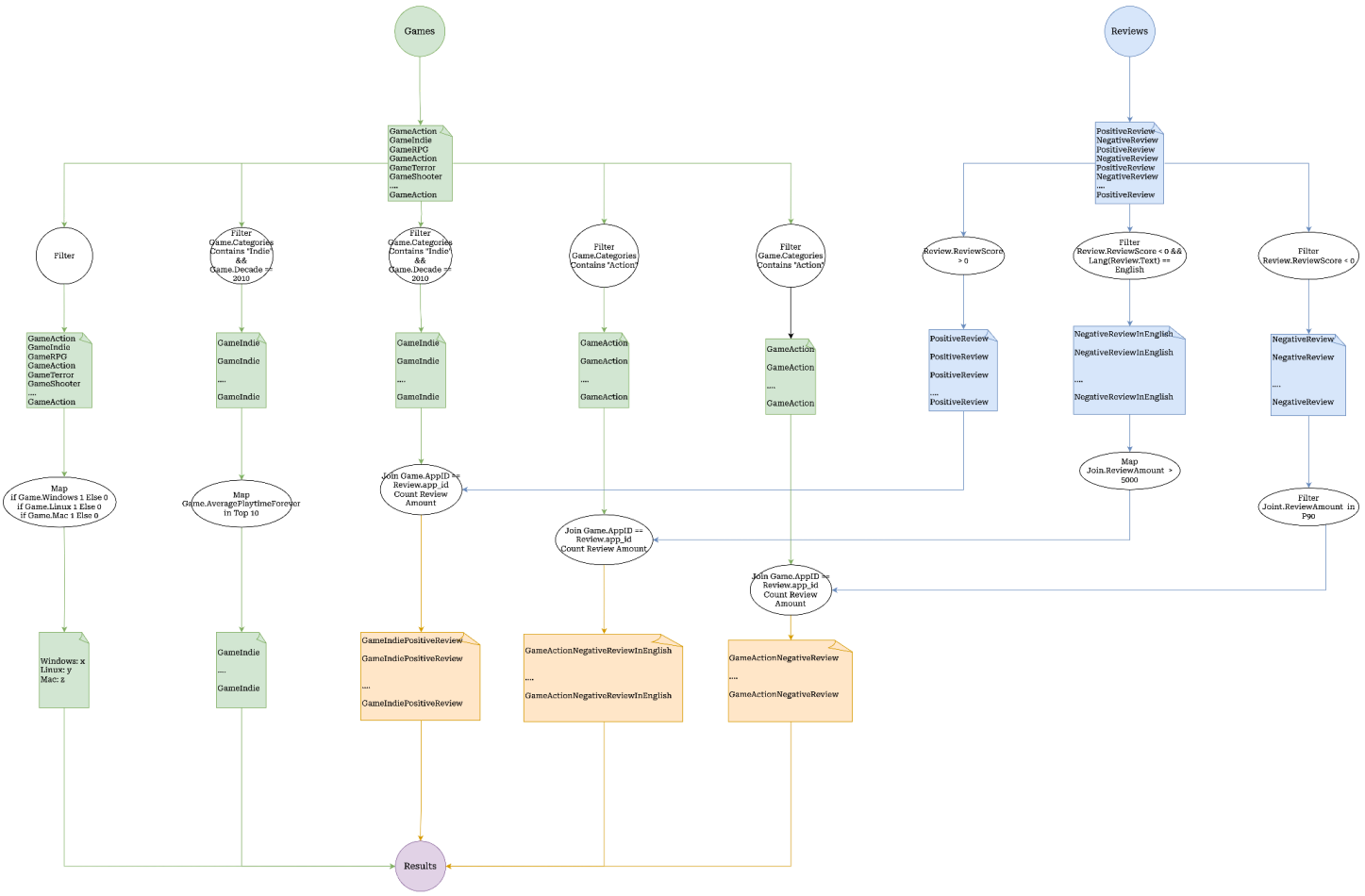
Vista de Procesos

Flujo del Sistema

El siguiente diagrama de flujo muestra el procesamiento de juegos y reseñas dentro del Sistema distribuido, enfocándose en cómo se aplican los filtros y las operaciones de unión, ordenamiento y agregación antes de obtener los resultados.

En general, el diagrama indica el flujo para alguna fila del dataset que le corresponda. Las operaciones de Join, Sort y P90 deben hacerse con la combinación de varias filas.

Flujo de Datos del Sistema



Diagramas de Secuencia

En el siguiente diagrama de secuencia se muestra como es el flujo de un mensaje recibido a través del trabajo de un *Controller* genérico, esta secuencia es independiente de la lógica de negocio del *Controller* en particular. La lógica de negocio solo debe preocuparse de poder conseguir el objeto necesario para realizar su trabajo a partir de un arreglo de bytes y de dar como resultado algo que pueda ser particionado y que pueda ser serializado.

El **handler** puede elegir si envía información hacia adelante luego de manejar un mensaje. En caso de mandar información hacia adelante, el flujo cambia ligeramente.

Diagrama de Secuencia de Recepción y Manejo de un Mensaje de Información

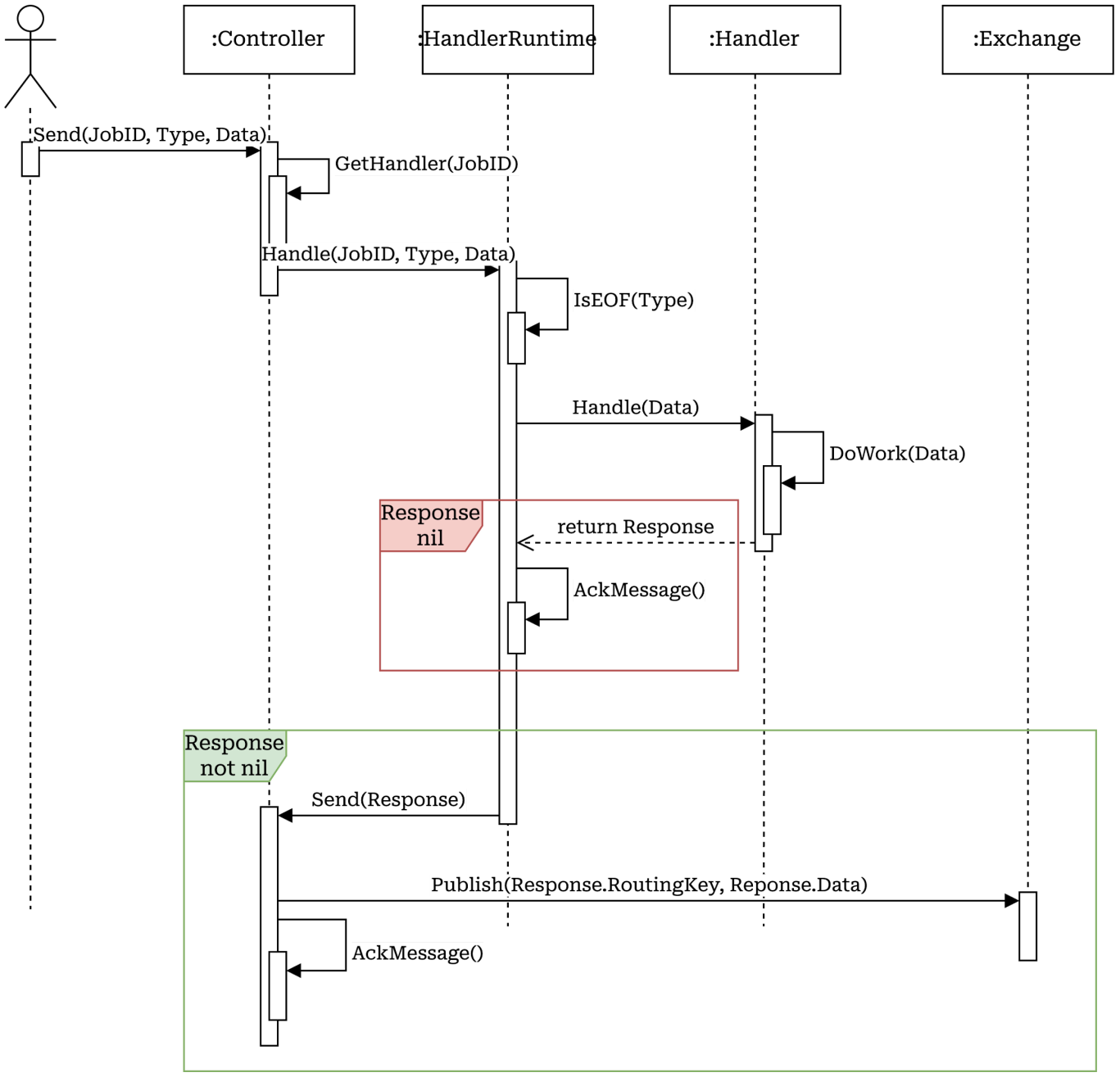
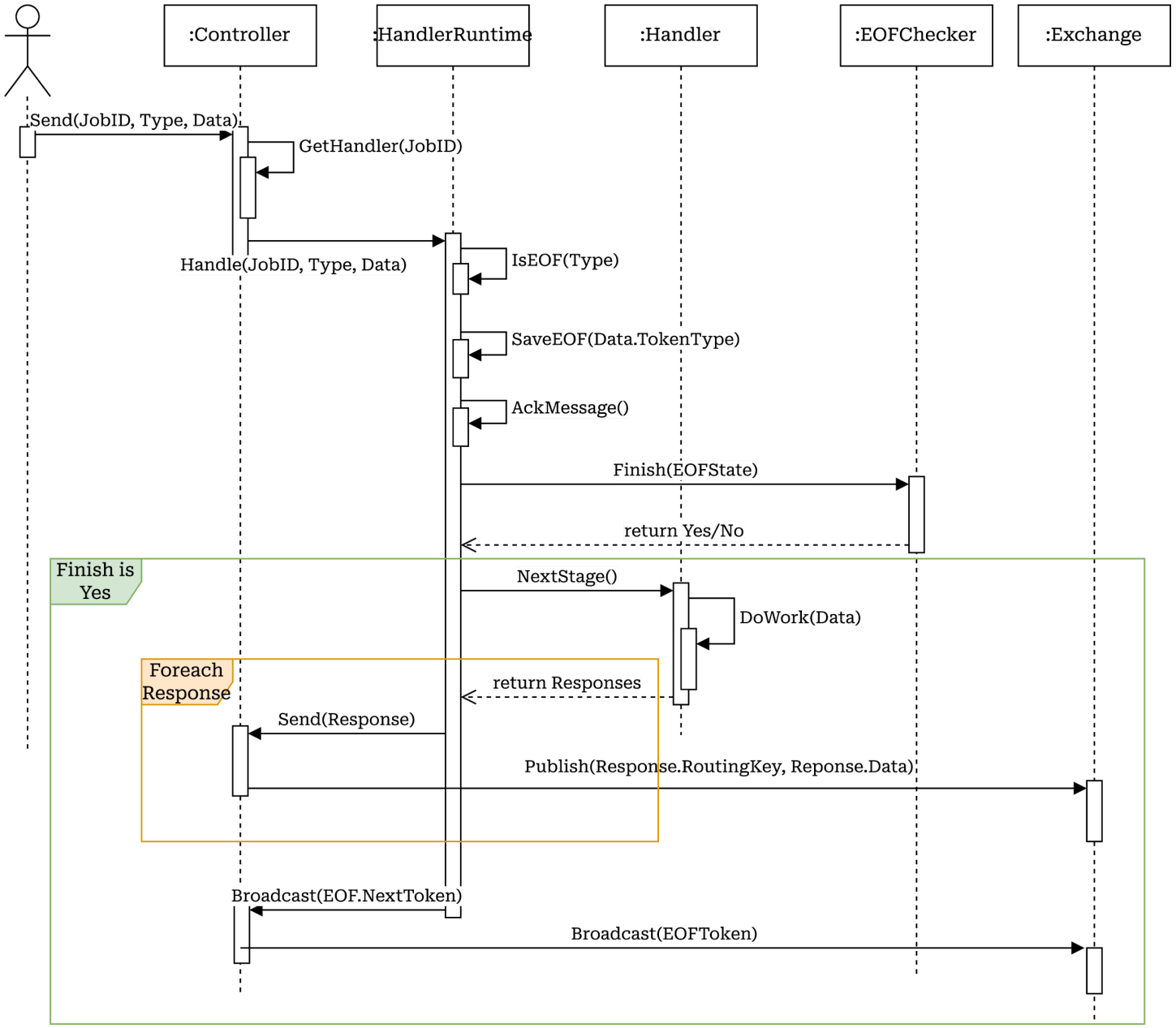


Diagrama de Secuencia de Recepción y Manejo de un Mensaje de EOF



Ahora bien, a grandes rasgos la secuencia general del Sistema interactuando para las distintas *queries* se verá como:

Diagrama de Secuencia General para la Lógica de Negocio

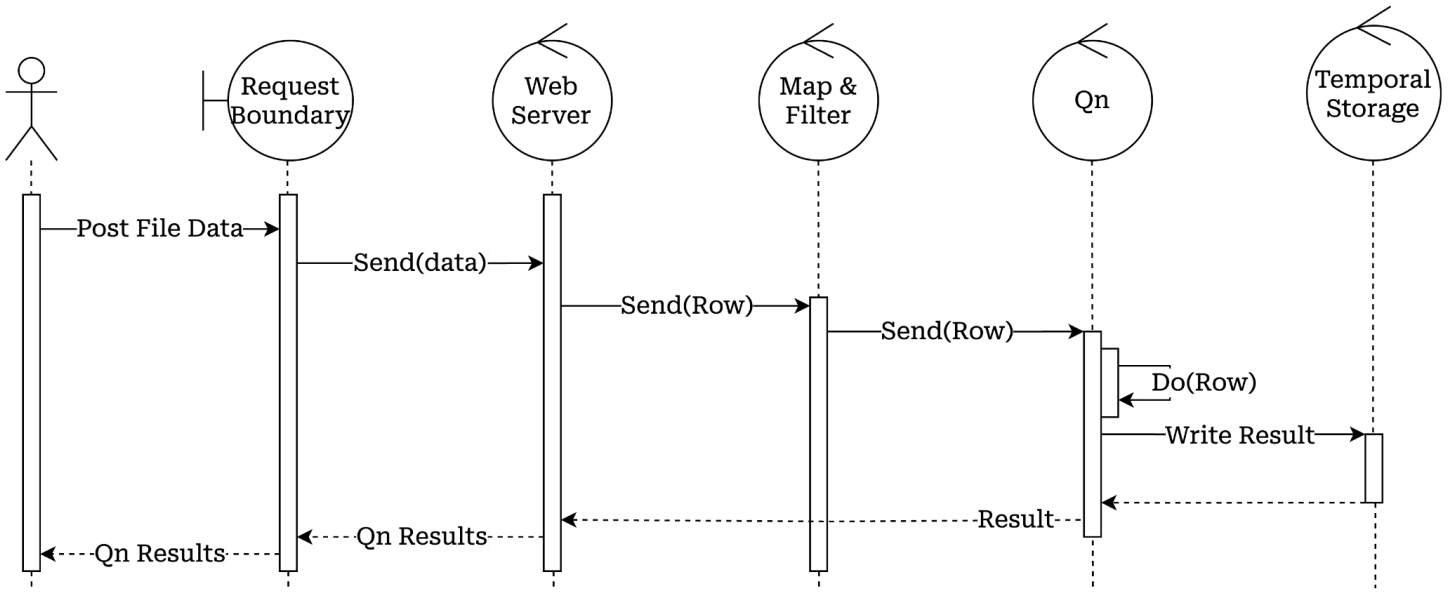
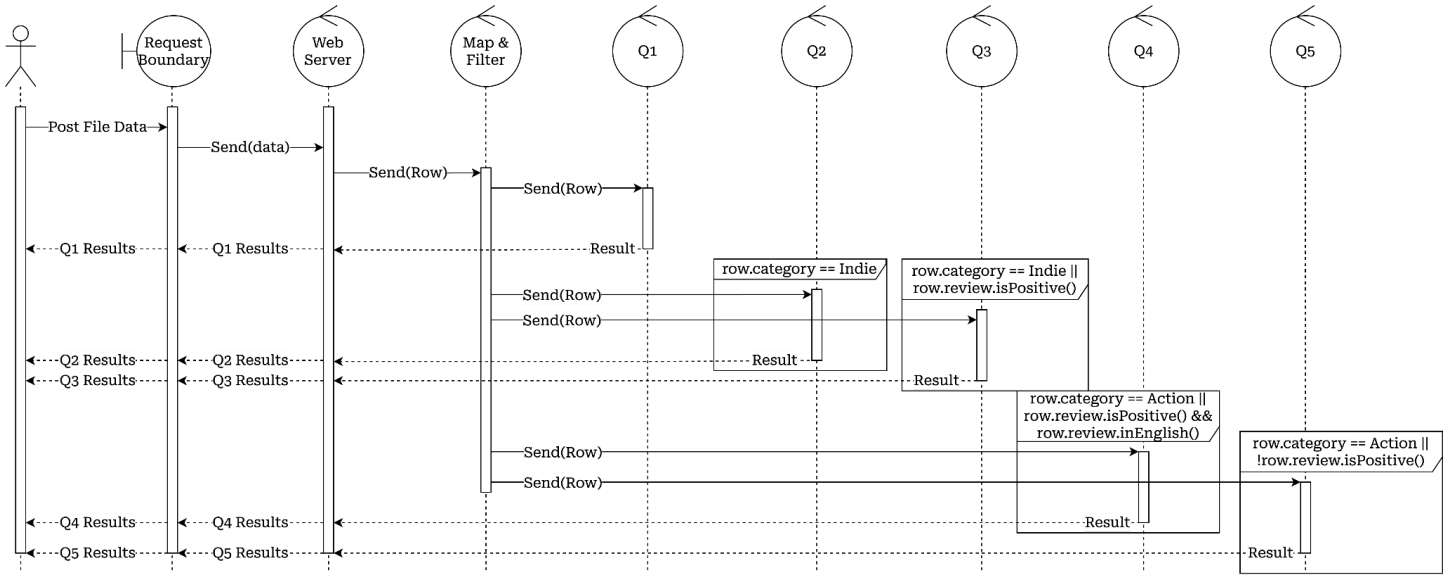


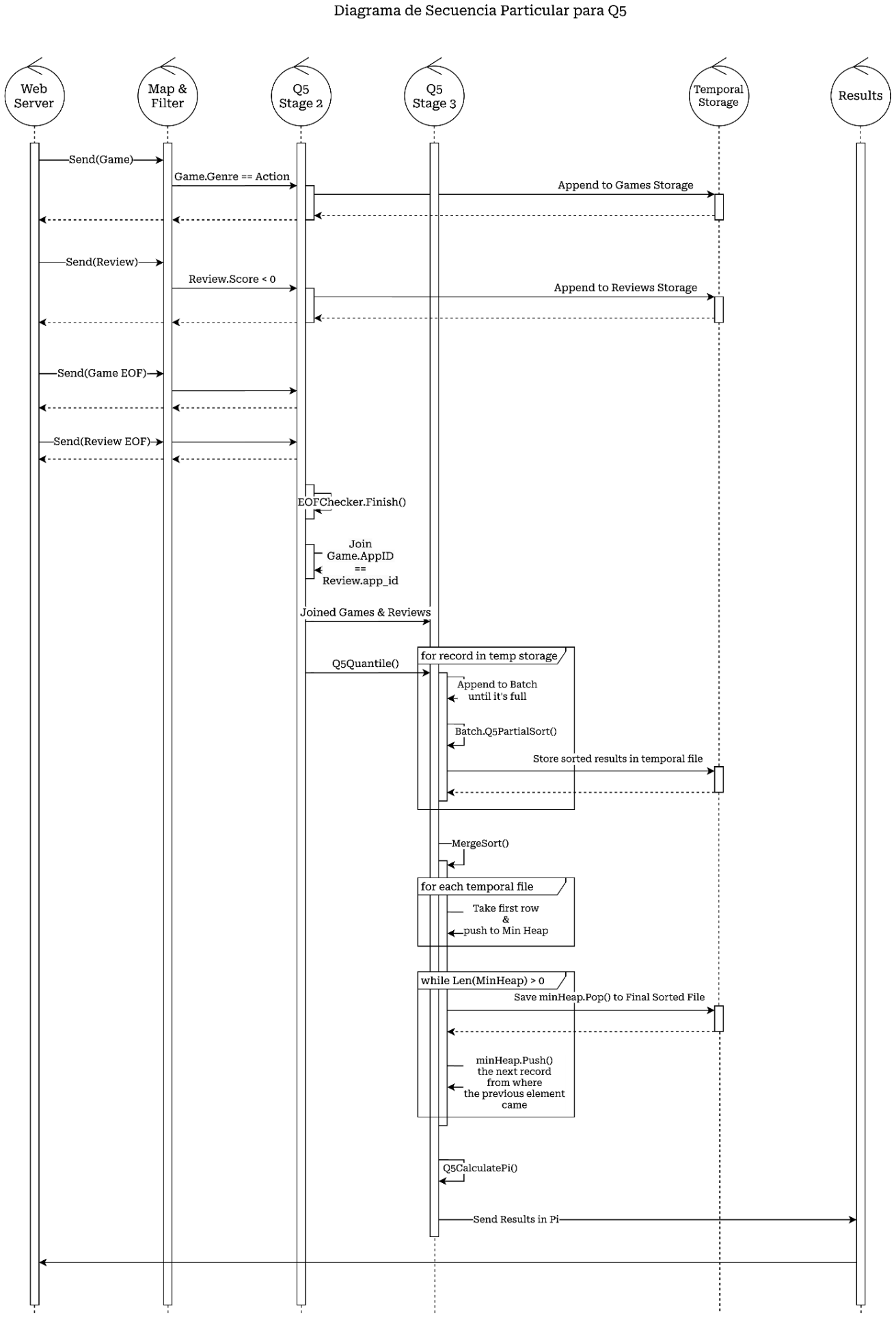
Diagrama de Secuencia General para la Lógica de Negocio: interacción de todas las Queries



Puesto que la lógica de negocio de la *query 5* es considerablemente más compleja en comparación al resto de las *queries*, adjuntamos a continuación un diagrama de secuencia que detalla el flujo con el cual se procesará la información de forma tal de devolver aquellos juegos que entran en el percentil i-ésimo (pues a fin de cuentas es un parámetro configurable).

Cabe destacar que el método utilizado para ordenar los registros es partiendo la información en distintos archivos temporales (con un criterio de corte de un peso máximo). Estos archivos serán parcialmente ordenados, para luego ejecutar un *MergeSort* que junte y ordene la información por completo. Como máximo, en memoria se ordenará la información que se almacenará en los archivos temporalmente ordenados, de forma tal de no saturar la memoria utilizada. El ordenamiento final es

realizado utilizando una estructura de Heap de Mínimos que mantiene, como máximo, un registro por cada archivo temporal creado.

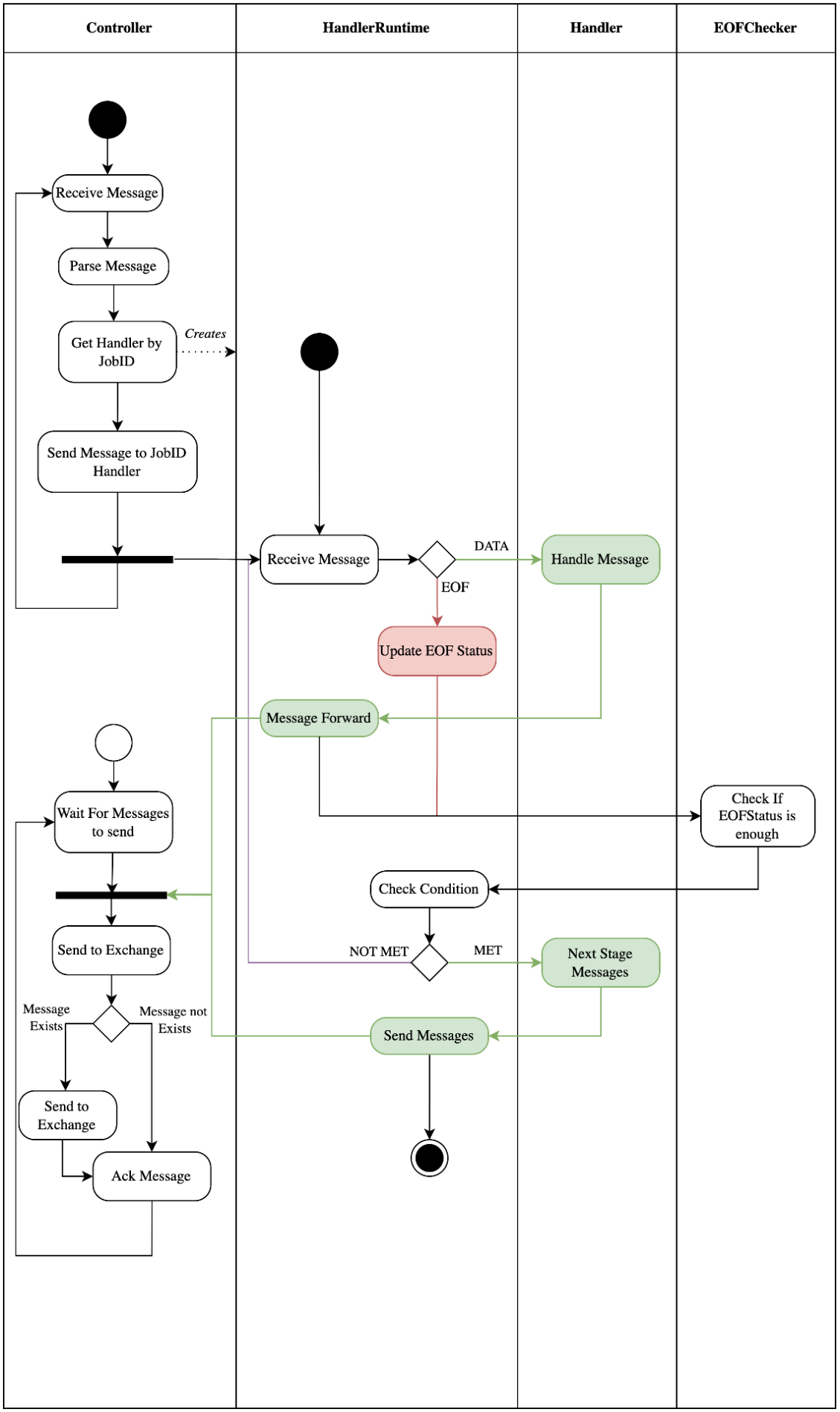


Diagramas de Actividades

A continuación enunciamos el diagrama de actividades que detalla el flujo central del *Controller* de un *worker* genérico. El *Controller* será uno por cada tarea que se necesite en el sistema, este puede ser un solo *Controller* por nodo, o varios en un mismo nodo, si la capacidad de cómputo lo permite.

En general, el *check* de la condición de *EOFs* necesarios para avanzar en el trabajo dependerá del tipo de *Controller* y la cantidad de particionado en la parte “posterior” del nodo (según el diagrama de robustez).

Diagrama de Actividades EOF

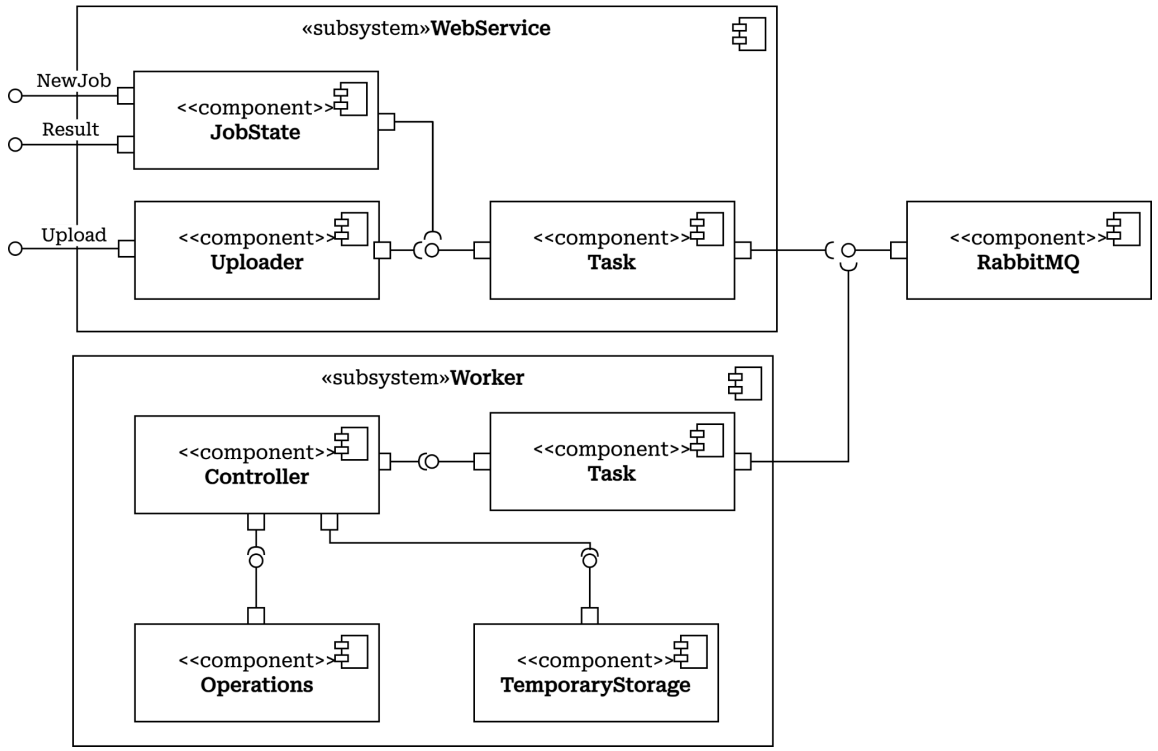


Vista de Desarrollo

Diagramas de Componentes

La comunicación entre los dos grandes subsistemas y entre los diferentes *workers* será realizada a través de mensajes en colas de RabbitMQ previamente definidas. Las interfaces entre los componentes del Sistema serán a nivel de lenguaje de programación (no se necesitan enviar mensajes por la red).

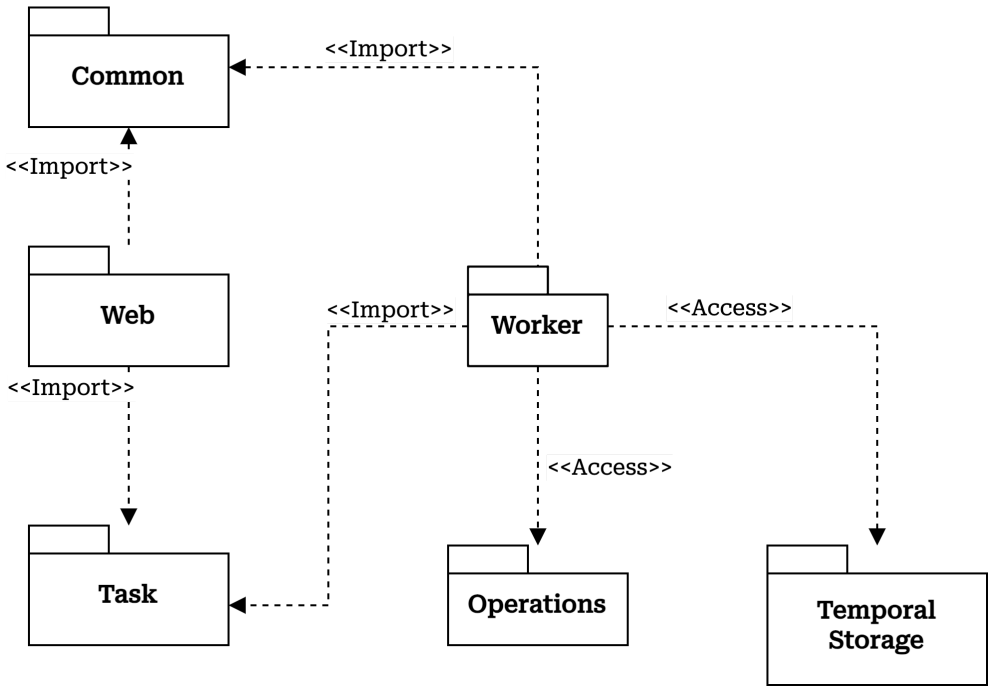
Diagrama de Componentes



Diagramas de Paquetes

El paquete `Common` tendrá funcionalidades genéricas para el correcto funcionamiento del programa. Esto puede ser desde leer parámetros de un archivo, manejo correcto de señales, etc.

Diagrama de Paquetes



Vista Física

Diagrama de Despliegue

En general existirán tres tipos de nodos distinguidos:

- `Web Server`
- `Worker`
- `Client`

En lo respecta a los `Worker` podemos ver que entre todos se mantiene una estructura común a pesar de la individualidad de cada una de sus actividades. En esta línea, cada nodo tendrá acceso a su propio almacenamiento interno que le permitirá realizar tareas de procesamiento y persistencia de datos.

Diagrama de Despliegue

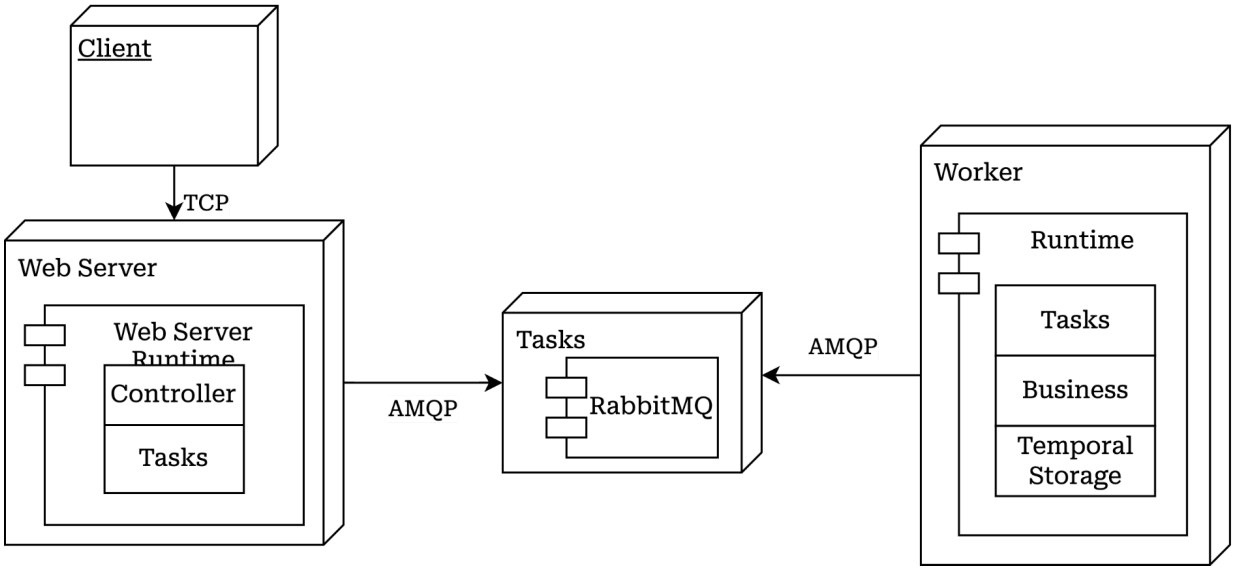


Diagrama de Robustez

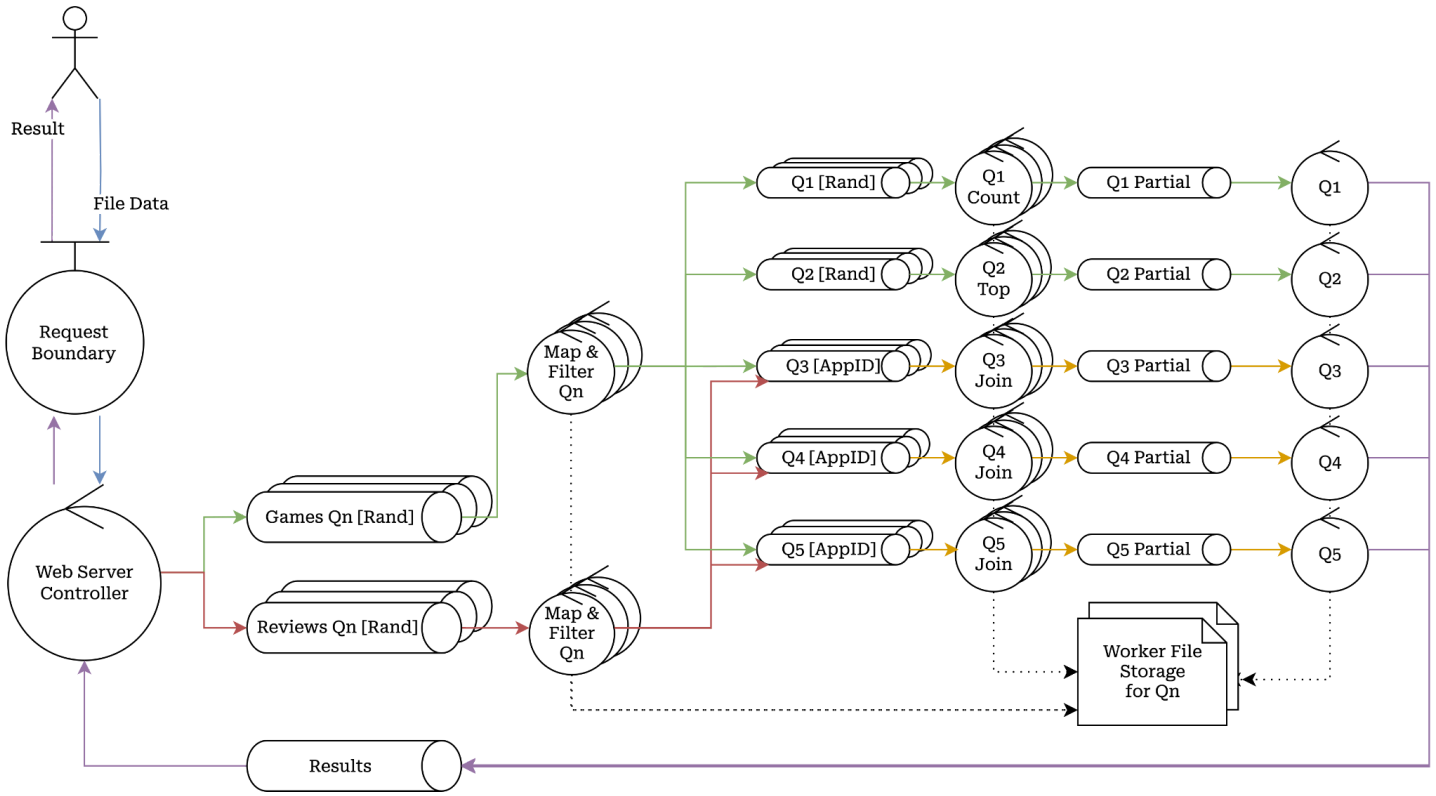
Cada `Worker`(Q_n) tendrá un *storage* dedicado para así poder tener persistencia de datos y no necesitar tener todo el estado en memoria. Por simplicidad, se evitó dibujar uno por cada *worker* y se graficó uno genérico.

Cada fila de cada dataset pasará por un worker que realizará el *Map & Filter* necesario para realizar la query y así poder tener la menor cantidad de datos necesarios a través de la red.

Luego, cada query pasará por una etapa intermedia, que se encargará de realizar una parte del trabajo para así poder escalar la cantidad de nodos en el sistema.

El Cliente deberá enviar el dataset asociado al JobID (previamente solicitado). Gracias a esto el sistema podrá trabajar con varios clientes a la vez.

Diagrama de Robustez



Atributos de Calidad

- **Scalability**: El Sistema debe ser capaz de escalar de forma tal que le sea posible manejar de manera controlada la demanda de los Clientes, quienes serán representados a través de nodos en el Sistema.
- **Reliability**: El Sistema deberá soportar caídas -en principio- aleatorias de los diferentes nodos y seguir funcionando correctamente.
- **Portability:** El Sistema deberá poder ser desplegado (*deployed*) de forma agnóstica a la cantidad física de procesadores, ya sea virtuales o físicos. El sistema deberá correr en sistemas compatibles con Linux.
- **Performance**: El Sistema debe ser capaz de procesar las consultas de manera eficiente, garantizando tiempos de respuesta aceptables incluso con grandes volúmenes de datos.
- **Usability**: El Sistema debe ser fácil de usar para los clientes, proporcionando interfaces claras y accesibles para enviar datasets y recibir resultados.