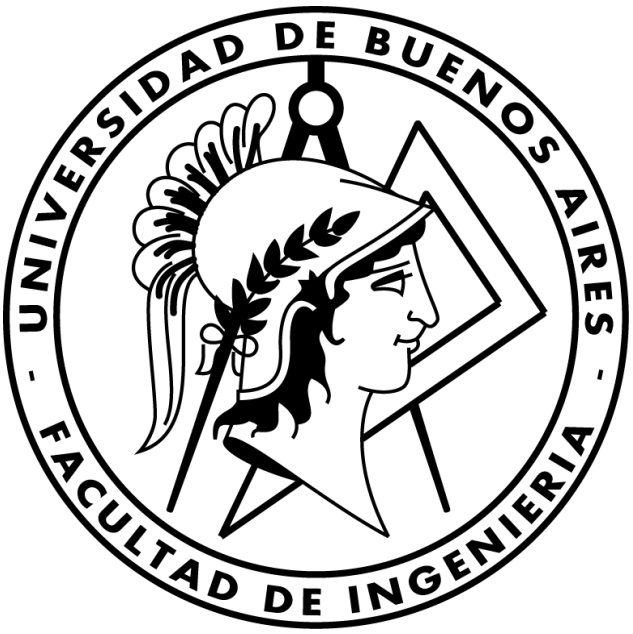


Trabajo Práctico Final

Sistemas Distribuidos I

Primer Cuatrimestre 2021

Integrantes:

Olivia Fernandez	99732
Cecilia Hortas	100687

Índice

Scope

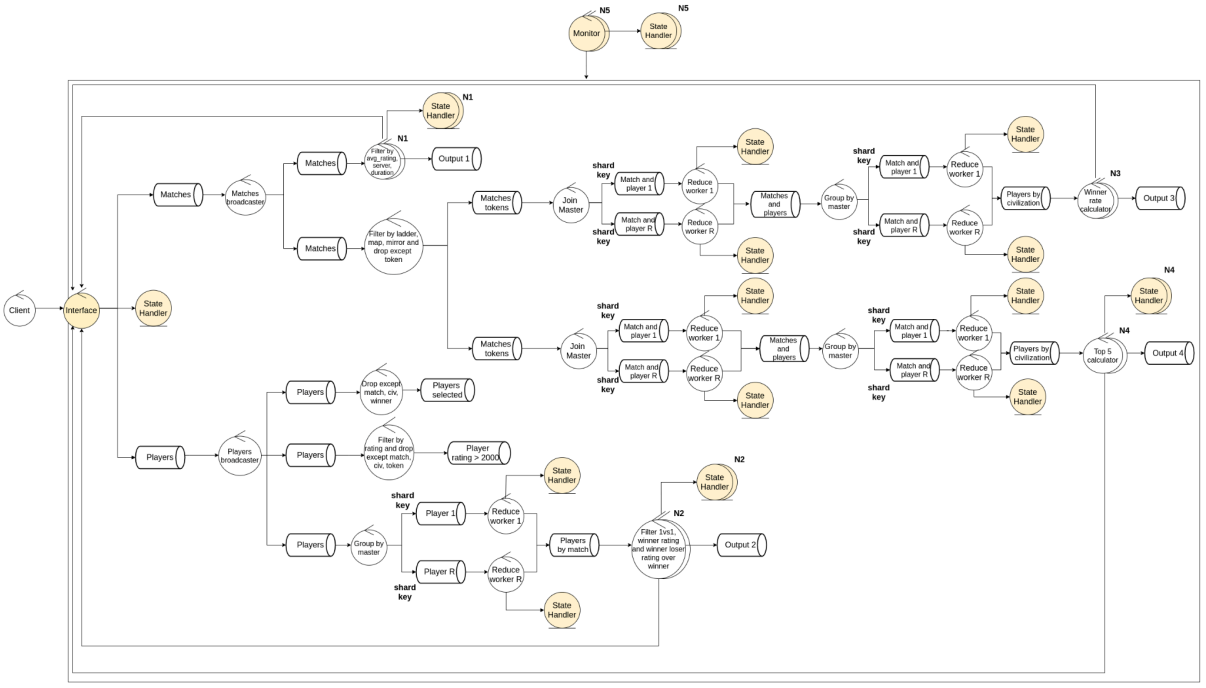
El objetivo de este trabajo es extender la funcionalidad del segundo trabajo práctico para soportar los siguientes requerimientos:

- El sistema debe mostrar alta disponibilidad hacia los clientes
- El sistema debe ser tolerante a fallos como la caída de procesos
- El sistema debe permitir procesar datasets secuencialmente

Para poder cumplir estos requerimientos se tuvieron que realizar cambios a la arquitectura que se expondrán en los diversos diagramas de este informe. Además de implementar un sistema que soporte estos requerimientos, es necesario justificar las decisiones de diseño tomadas con sus respectivas ventajas y desventajas, teniendo en cuenta además que una variable importante en tales decisiones es que el tiempo de desarrollo es acotado.

Vista física

Diagrama de robustez



Se esquematizan en el diagrama los distintos nodos del sistema y cómo se comunican entre sí. Los nodos que se ven representados con fondo blanco son los mismos que se usaron en el segundo trabajo práctico así que no se explicará su responsabilidad en este informe. Por otro lado, los nodos amarillos son los que se tuvieron que agregar para manejar los nuevos requerimientos del sistema y se procede a describir su responsabilidad:

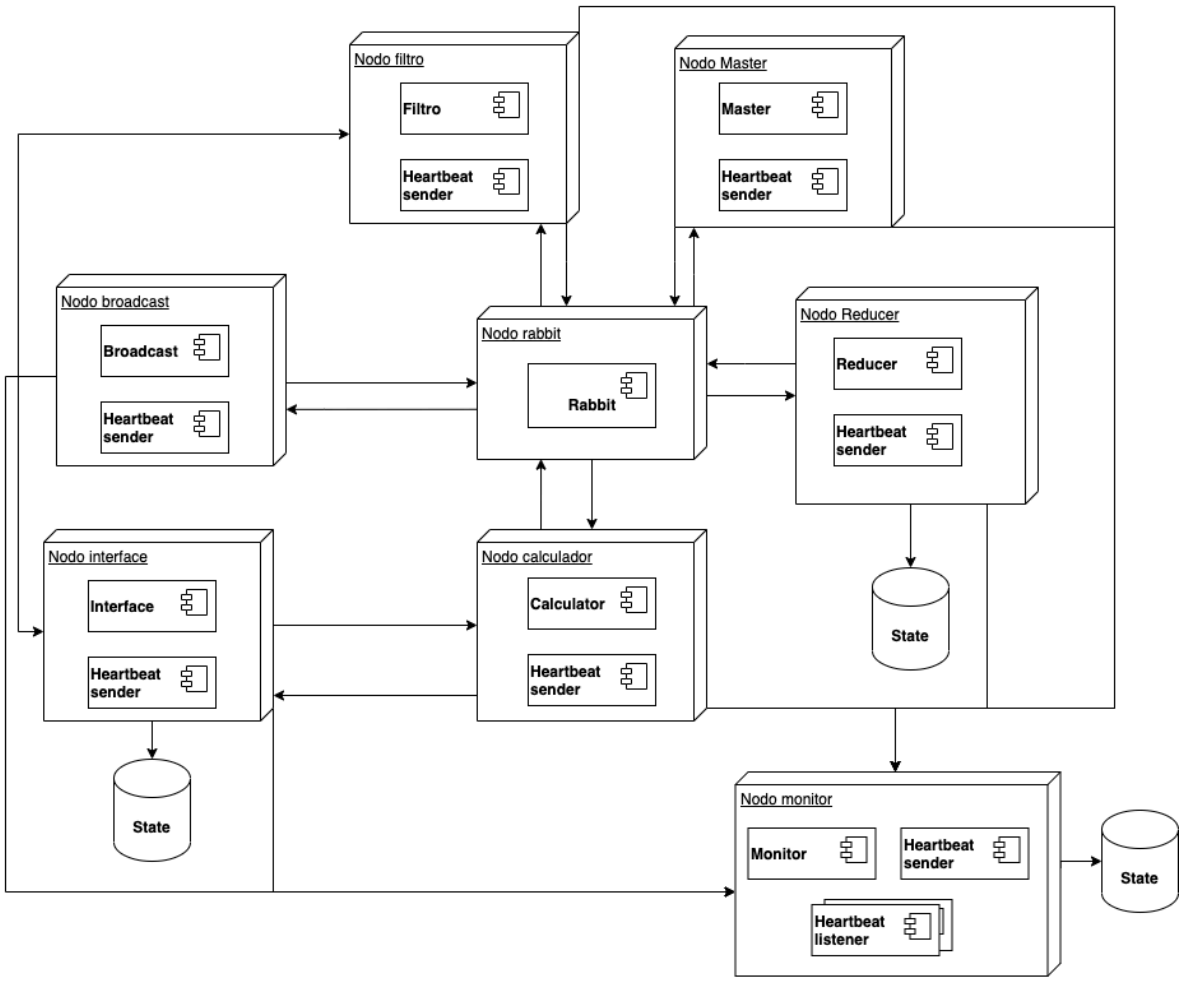
- Interface: Es el nodo encargado de comunicarse con el cliente para procesar la query. Como ahora los datasets se procesan secuencialmente, se tuvo que agregar

este nodo para dar una respuesta al cliente tanto para el caso que el sistema pueda procesar la query como para el caso en el que está ocupado resolviendo una query previa.

- Monitor: Es el nodo encargado de comunicarse con todos los nodos del sistema para detectar fallas de los mismos. En el caso en que detecte que un nodo está caído se encarga de levantarlo.
- StateHandler: Se encarga de persistir los datos de los distintos nodos stateful, es decir, que requieren estado. Si un nodo se cae por algún motivo, cuando se levante debe tener su estado previo, por ejemplo conocer qué datos procesó previamente para que el resultado de la query no se vea afectado. Si bien no es un nodo en sí mismo se agregó al diagrama para remarcar los nodos que requieren estado.

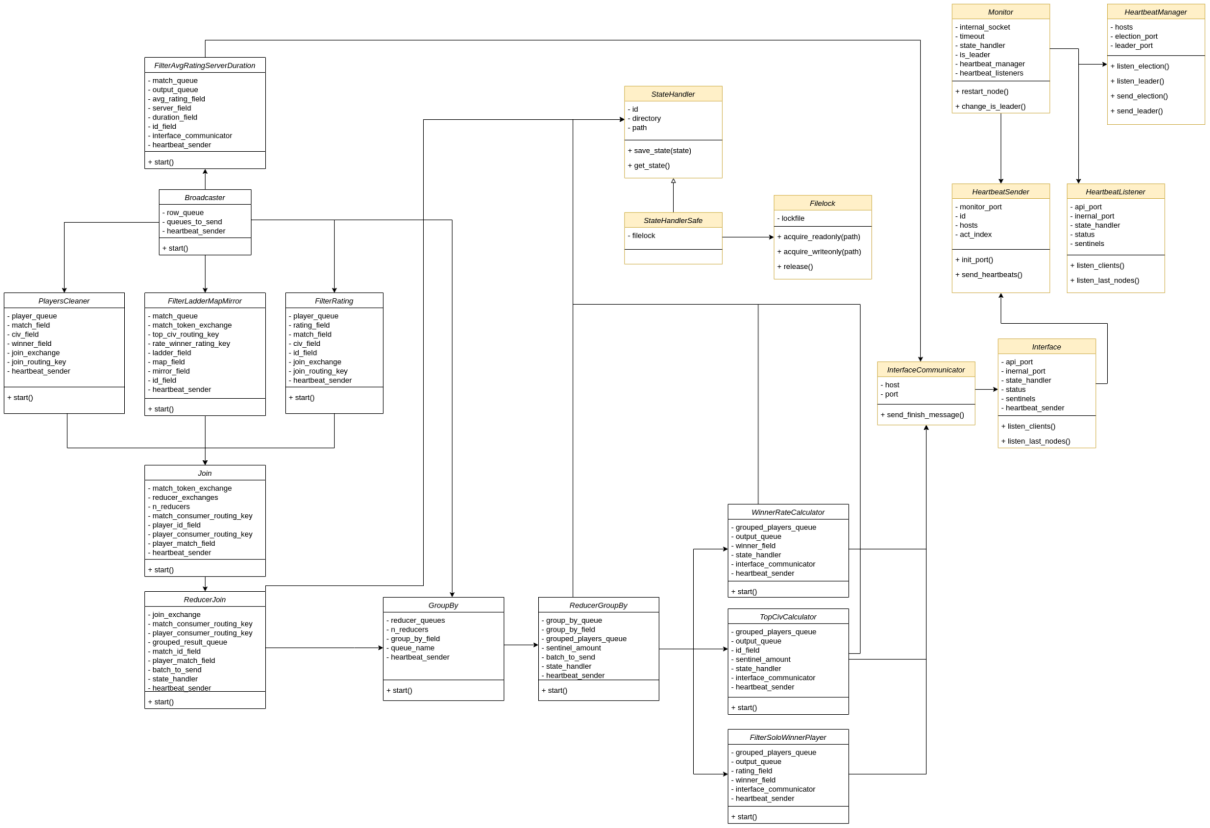
Diagrama de despliegue

A continuación se presenta el diagrama de despliegue del sistema. Se simplifican los nombres de los distintos nodos ya que guardan semejanza con los del diagrama de robustez. Se busca mostrar en este diagrama qué nodos constituyen el sistema y cuáles son los que ingresan la entrada, consumen la salida y guardan estado.



Vista lógica

Diagrama de clases



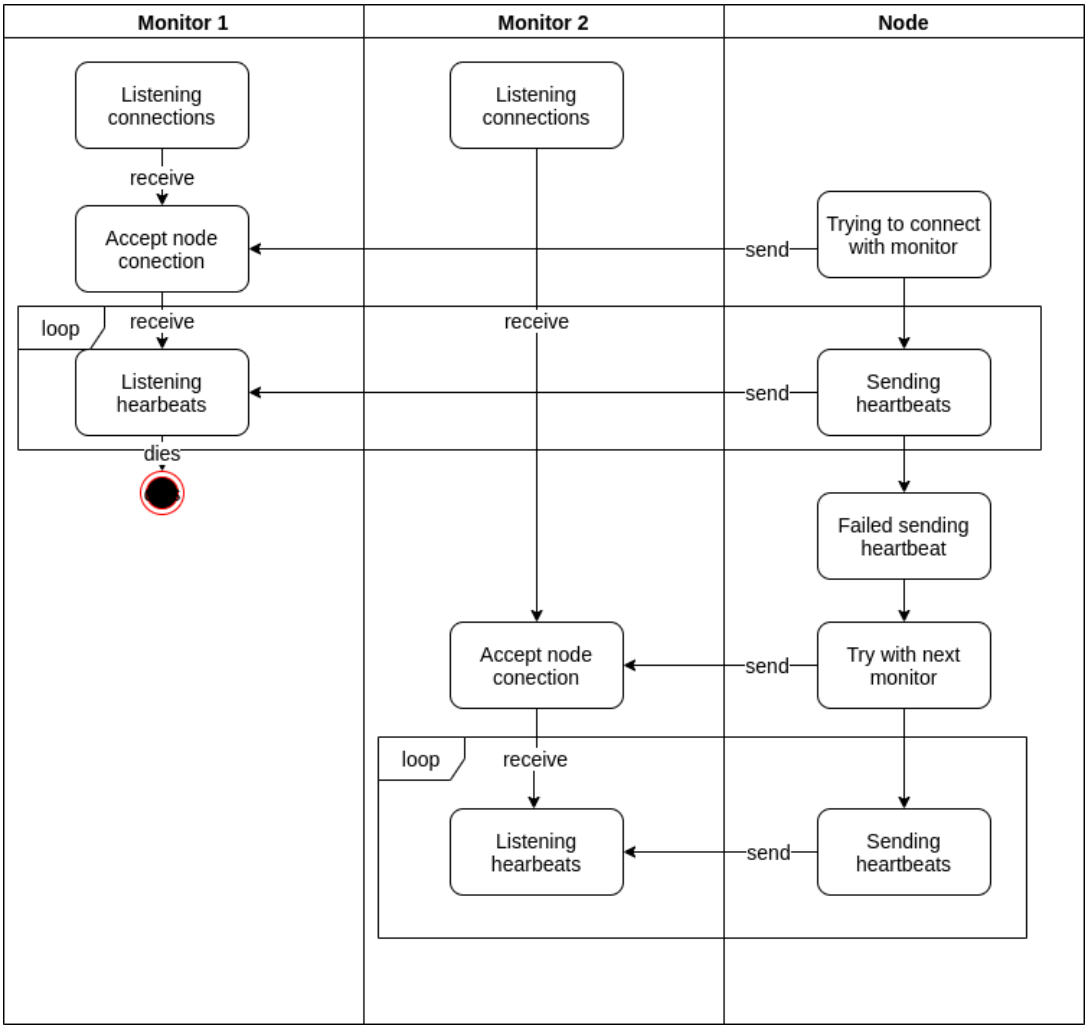
Al igual que en el diagrama de robustez se observan en amarillo las principales clases que se tuvieron que agregar en este trabajo práctico. Se procede a detallar a grandes rasgos qué responsabilidad tiene cada clase:

- StateHandler: se encarga de manejar el estado de las clases stateful para evitar que la caída de un nodo implique pérdida de información de procesamiento
- StateHandlerSafe: se creó para poder acceder y modificar concurrentemente el estado de un nodo desde distintos threads de manera segura
- Filelock: utilizado para poder acceder a un archivo de manera concurrente
- InterfaceCommunicator: clase donde se encapsula la conexión con la interfaz para avisarle que se terminó de procesar el dataset, ésta entidad la poseen los últimos nodos de cada pipeline.
- Interface: se encarga de comunicarse con el cliente para manejar sus requests
- Monitor: se encarga de monitorear a los nodos que se suscriban a él para poder detectar cuándo fallan y levantarlos en caso de ser necesario
- HeartbeatSender: se encarga de conectarse a un nodo por medio de sockets y mandar heartbeats cada cierto periodo de tiempo. Si llega a fallar el envío, se suscribe al próximo nodo y le envía heartbeats
- HeartbeatListener: se encarga de escuchar heartbeats de los nodos que estén suscritos y en caso de que expire un cierto timeout configurable, se levanta al nodo.
- HeartbeatManager: implementa el algoritmo de elección de líder y maneja el envío de heartbeats entre monitores

Vista de procesos

Diagrama de actividad

Diagrama de actividad del envío de heartbeats



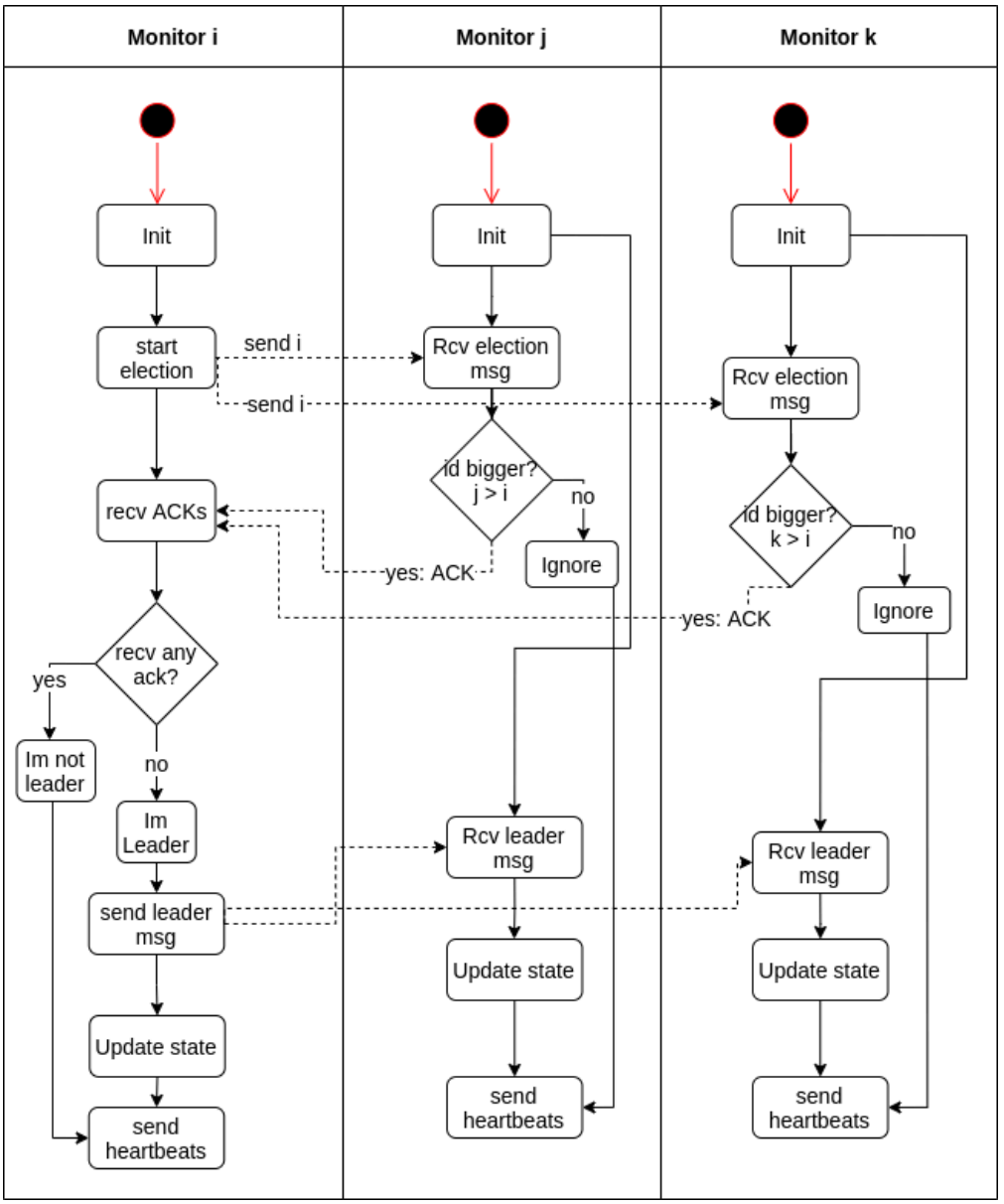
A fin de garantizar alta disponibilidad y además de que el sistema sea tolerante a fallos, fue necesario implementar algún tipo de mecanismo para que los monitores del sistema detecten si un nodo se cayó y así poder levantarlo nuevamente con *docker in docker*. Para ello se debatieron dos alternativas:

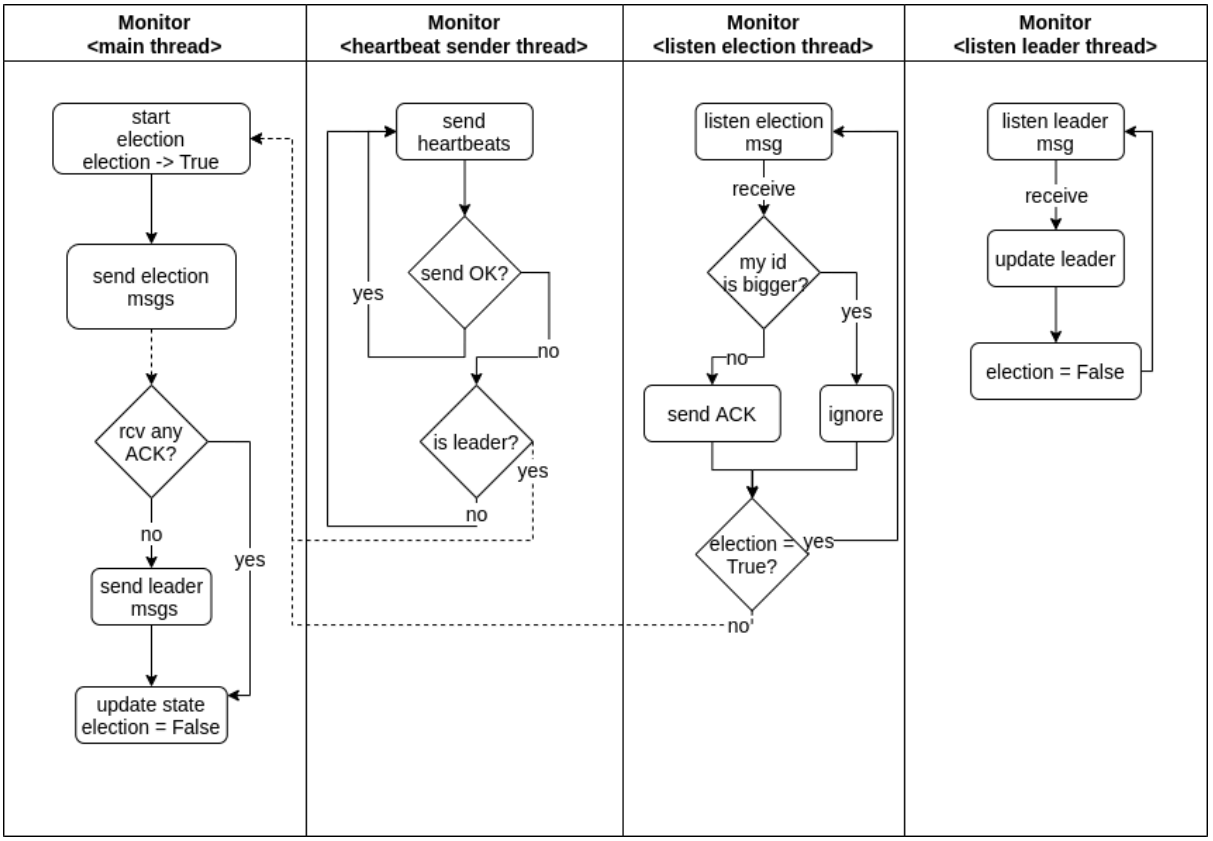
- Que cada nodo envíe un heartbeat a un monitor designado
- Que el monitor pinguee a los nodos que le correspondan

Se decidió elegir la alternativa del hearbeat debido a que implicaba el envío de un solo mensaje (del nodo hacia al monitor), a diferencia de la alternativa del ping que implicaba el envío de dos mensajes (un ping desde el monitor hacia el nodo y una respuesta del nodo hacia el monitor). Además, con la implementación actual el monitor no tiene que conocer a los nodos, sino que los nodos se suscriben al monitor que corresponda para ser monitoreado, lo cual consiste en una gran ventaja ya que se pueden agregar nuevos nodos al sistema y el monitor no debe cambiar su configuración para poder monitorearlos.

Una vez elegido el mecanismo de monitoreo se debió decidir cómo se iba a implementar este mecanismo de envío de heartbeats. Para ello nuevamente hay varias alternativas, por ejemplo una implicaba la designación de un líder y que dicho líder fuera el encargado de detectar las fallas de los nodos. Sin embargo, no se eligió esta alternativa ya que una implementación más sencilla cumpliría el requerimiento. Esta alternativa consiste en que cada nodo conoce las ips de todos los monitores por lo que se puede conectar a cualquiera de los mismos con un mecanismo de suscripción y ante la detección de la caída de dicho monitor, pasa a suscribirse al próximo monitor en la lista y así sucesivamente, en un estilo *round robin.* Esta alternativa tiene la ventaja de que no es necesario implementar un algoritmo de elección de líder junto con las dificultades que esto conlleva y además que, si bien la carga del sistema no es muy alta, se puede distribuir entre los distintos monitores, ya que todos los monitores podrían estar escuchando heartbeats, a diferencia de la implementación del líder donde solo él está haciendo el trabajo. Sin embargo, es pertinente destacar un punto débil de esta implementación que radica en que si el monitor al que está suscripto el nodo y el nodo se caen a la vez, el nodo nunca se va a recuperar ya que los monitores se encargan de detectar al nodo caído pero el monitor no guarda los nodos que estaban suscriptos a él por lo que no se dará cuenta que el nodo cayó en su ausencia. Una solución a esto que no pudo ser implementada por falta de tiempo es que los monitores guarden en su estado qué nodos están conectados a él en cada momento. De esta manera, cuando se levante el monitor, podrá detectar que dicho nodo no está enviando heartbeats y se encargará de levantarlo.

Diagramas de actividad del algoritmo de líder para los monitores

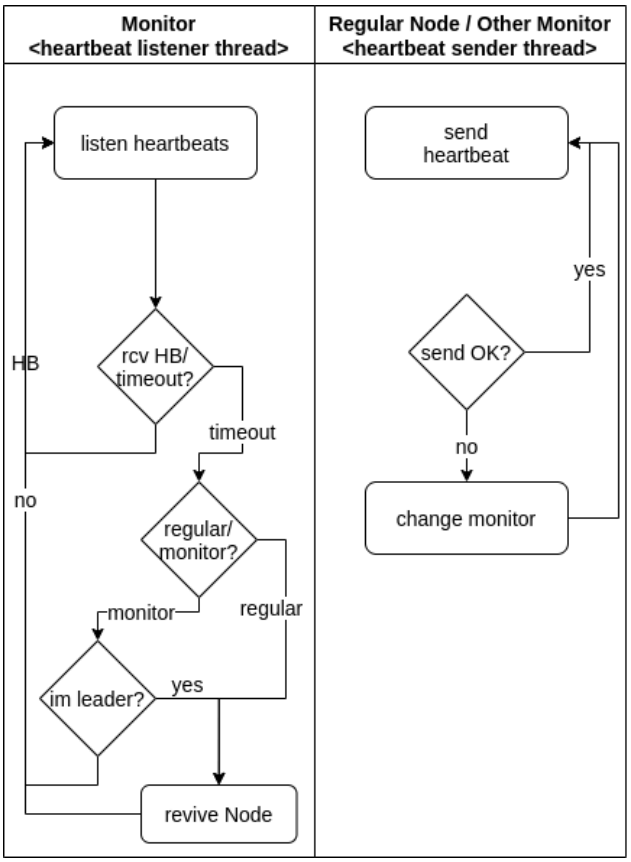




El mecanismo de detección de fallas para los monitores es distinto al explicado anteriormente para detectar las fallas de los nodos regulares del sistema. Se tiene un monitor líder que es el encargado de levantar al nodo que se haya caído y además todos los monitores se envían heartbeats entre sí. En caso de que se caiga un monitor que no es el líder, el líder se encarga de levantarlo. En cambio, si se cae el líder cualquiera de los otros monitores del sistema pueden detectarlo y desatar una elección. Para eso se tuvo que elegir un algoritmo de elección de líder: el algoritmo bully. Dicho algoritmo establece que el proceso con id más grande entre los procesos que no hayan fallado es el que se elige como líder. Una vez elegido el nuevo líder este se va a encargar de levantar al nodo que cayó previamente y luego desatar nuevamente una elección para que dicho monitor sea el líder.

Como fue explicado previamente, la desventaja de este enfoque fue la complejidad que agregó al sistema desarrollar un algoritmo de elección. Pero no se destaca alguna desventaja significativa, ya que si bien todos los nodos se envían heartbeats entre sí y eso agrega carga al sistema, se supone que la cantidad de monitores no va a ser tan alta como para que el tráfico de mensajes sea apreciable.

Diagramas de actividad de la implementación del monitor para levantar nodos

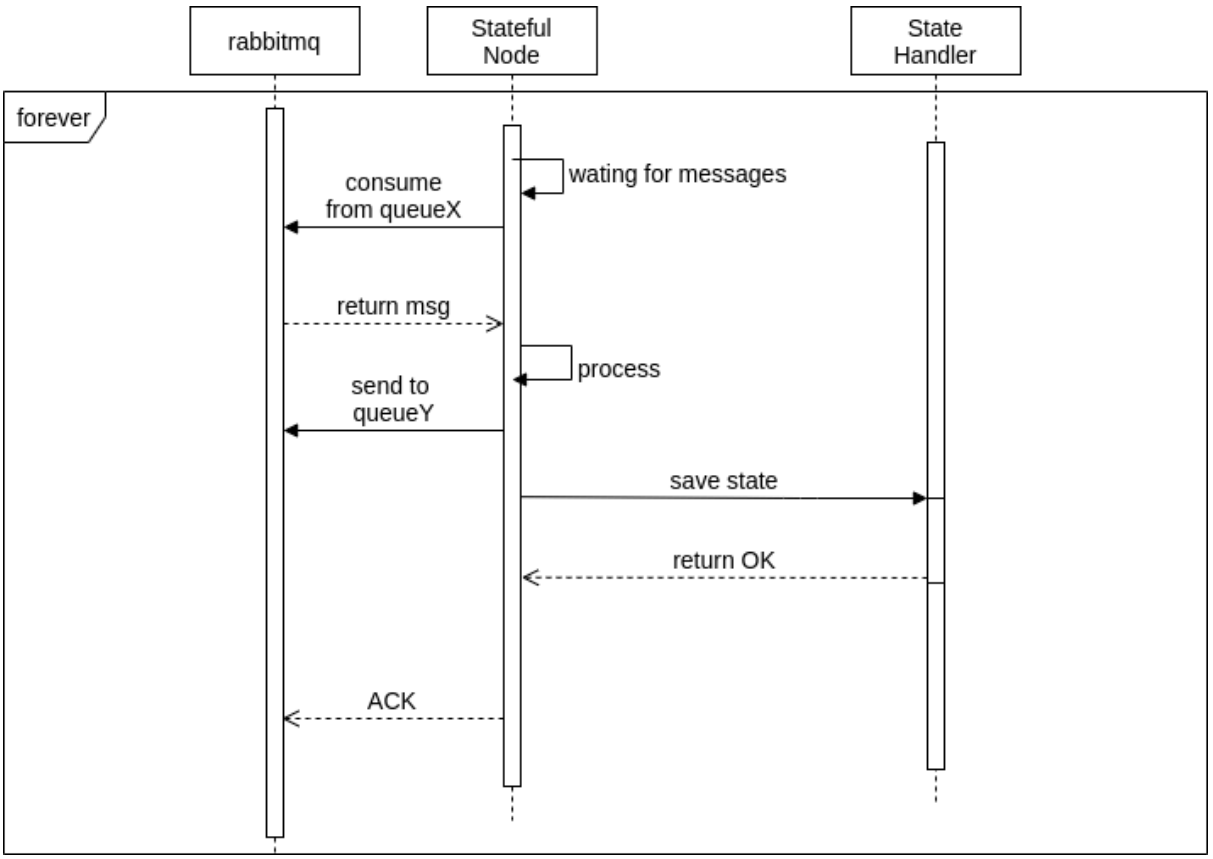


En este diagrama se expone cómo reacciona el monitor ante la caída de un nodo, sea un monitor o cualquier otro nodo. En el caso de que detecte que el nodo que se cayó es un nodo regular, lo levanta y sigue su ejecución. En caso de que sea un monitor el que se cayó (que no es el líder), solo se encarga de levantarlo nuevamente en caso de que el monitor que está monitoreando sea un líder. No se expone el escenario en el que se cae un líder ya que eso se mostró en los diagramas anteriores con la elección.

En la columna derecha se muestra el funcionamiento de envío de heartbeats de los nodos. Como fue explicado previamente, en caso que se detecte una falla en el envío se suscribe al monitor siguiente y en caso que la ejecución siga correctamente se repite el procesamiento otra vez.

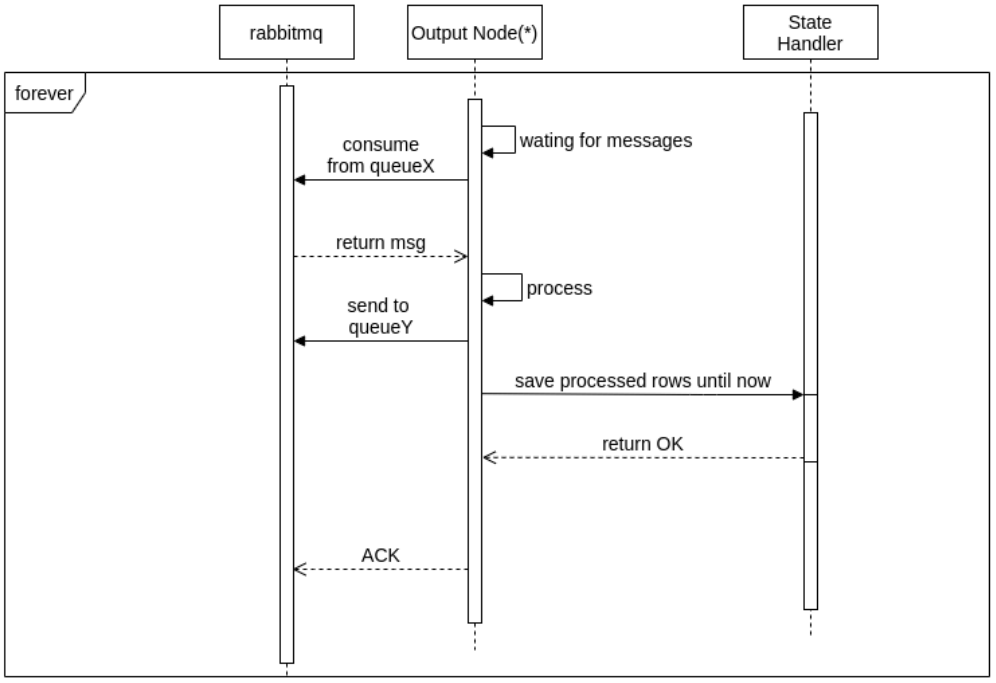
Diagrama de secuencia

Diagrama de secuencia para el estado de nodos stateful



Se expone el diagrama de secuencia para la operación de guardar estado en un nodo *stateful*. Se puede observar que cuando el nodo consume un mensaje y cambia su estado, se persiste y luego se envía el ack para poder consumir otro mensaje. Esta es una gran diferencia respecto al segundo trabajo práctico, donde al consumir cada elemento de la cola de rabbit el ack se hacía automáticamente. Acá fue necesario desacoplar el acto de consumir un mensaje y el de enviar un ack ya que si el nodo se cae previo a guardar su estado esa fila procesada se pierde.

Diagrama de secuencia para el manejo de duplicados

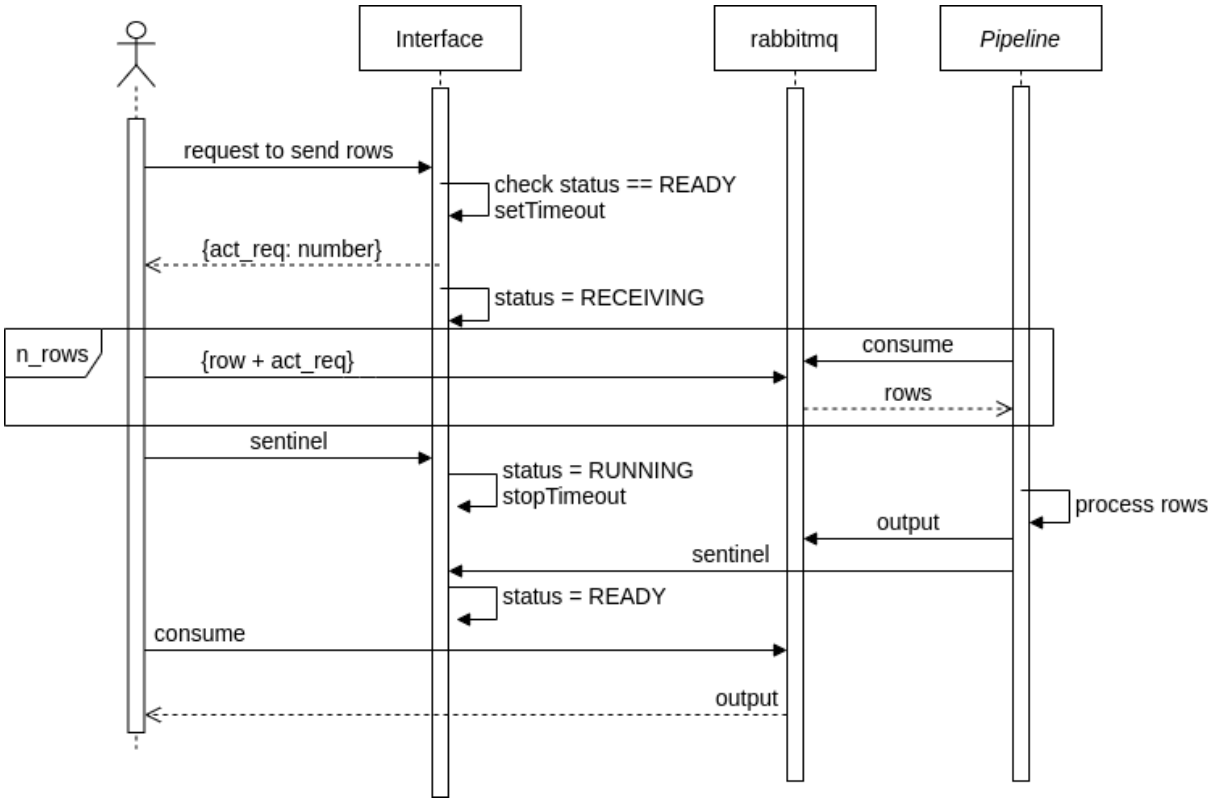


(*) Output node ∈ {Filter Avg Rating Server Duration, Filter Solo Winner Player, Winner Rate Calculator, Top Civ Calculator}

Se expone un diagrama muy similar al anterior pero esta vez para evidenciar cómo se manejan los duplicados en el sistema. Esto tuvo que realizarse debido a que un escenario posible de intercambio de mensajes implicaba la pérdida de un ACK, lo cual provoca que se reenvíe el mensaje en cuestión y así sea posible que se procese un duplicado en algún nodo.

Como se puede observar, los duplicados se controlan en los últimos nodos del sistema, es decir, en los nodos que escriben el resultado a las colas de salida. Una consecuencia directa de esto es que se podrían llegar a propagar datos de más por el resto de los nodos del sistema en caso de que se produzcan las pérdidas de los ACKs mencionados anteriormente. Sin embargo, se eligió esta alternativa ya que este escenario no es muy probable en las pruebas que se realizan al sistema, por lo que no circularía una carga apreciable de duplicados y además de la otra forma era necesario agregar estado en nodos que no lo tenían previamente, lo cual implicaba un procesamiento adicional que no valdría el esfuerzo. Con la implementación actual, cada vez que los últimos nodos reciben una nueva fila se chequea en el estado que no esté dentro de los ya procesados, es decir, que sea un duplicado, y de esta manera el resultado de la query es el mismo ante una caída del sistema.

Diagrama de secuencia para la interfaz



En este diagrama se presentan las principales acciones que realiza la interfaz ante la llegada de una request del cliente.

Es pertinente aclarar que este escenario ocurre si la interfaz no está procesando ninguna request. En caso contrario, la interfaz rechaza la request del cliente.

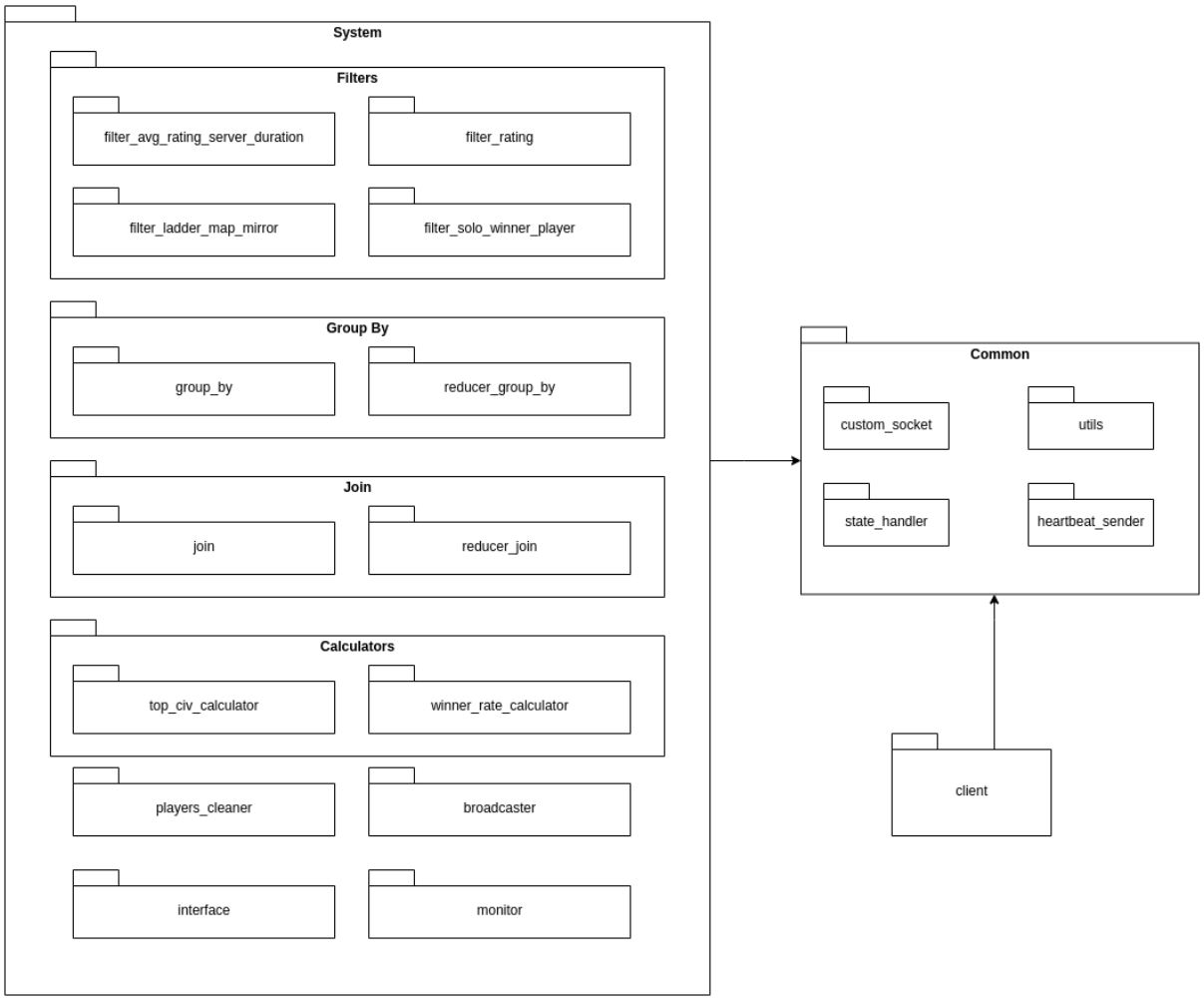
Se plantean los distintos estados para modelar la interfaz:

- READY: no está procesando ninguna request actualmente
- RECEIVING: el cliente está mandando filas al sistema y la query está corriendo
- RUNNING: el cliente dejó de mandar filas al sistema y la query está corriendo

El estado READY se plantea para que la interfaz sepa si el sistema está listo para correr una nueva query. Luego se plantea el estado RECEIVING para detectar si el cliente se cayó durante el envío de filas al sistema. Si detecta que se cayó, se pasa al estado READY para recibir otra request. En caso de que logre enviar todas las filas se pasa a estado RUNNING que significa que el sistema está corriendo la query. Una vez que los cuatro últimos nodos del sistema de cada query se comunican con la interfaz para anunciar que terminaron, el sistema pasa a estado READY de vuelta para aceptar otra query. De esta manera fue posible cumplir el requerimiento de procesamiento de datasets de forma secuencial. Además, con el uso del estado RECEIVING se contempla el requerimiento adicional de soportar la caída del cliente.

Vista de desarrollo

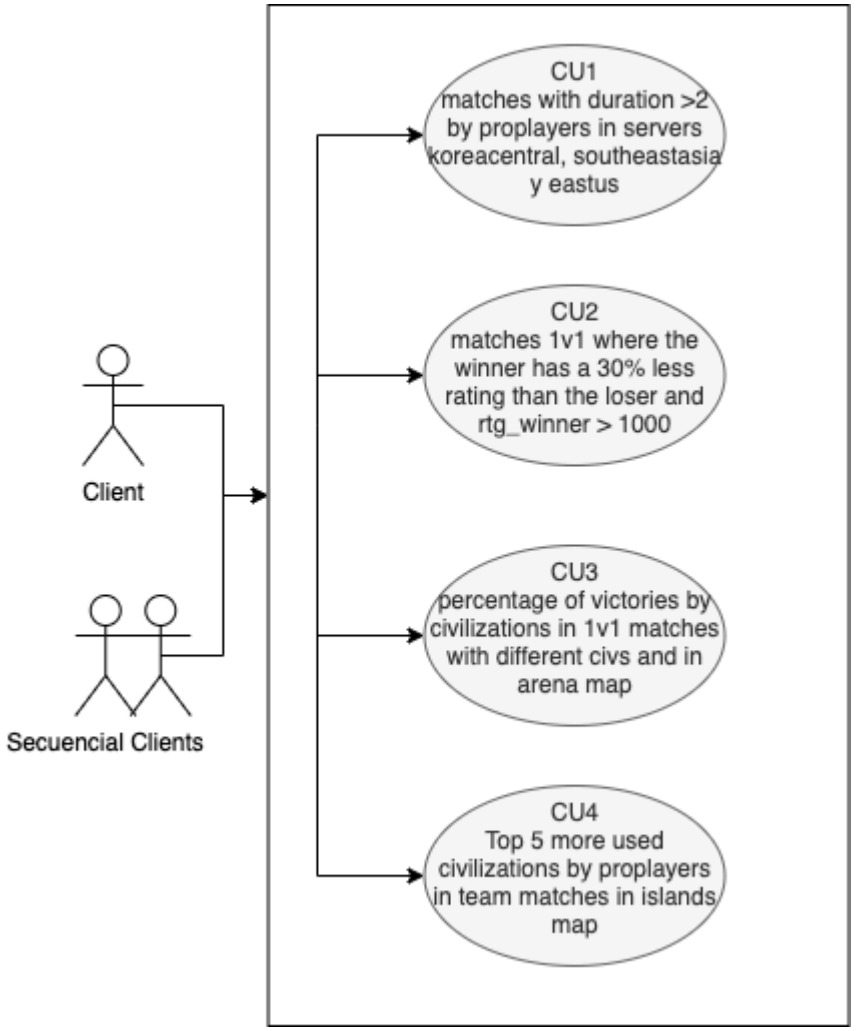
Diagrama de paquetes



Se expone el diagrama de paquetes para evidenciar la agrupación de los elementos UML del sistema que están relacionados. Lo que se busca mostrar es el uso de un paquete externo llamado common que guarda las distintas funciones de rabbit que se reutilizaron a lo largo del sistema.

Escenarios

Casos de Uso



Se presenta el diagrama de casos de uso del sistema. Se soportan cuatro querys por request y además se procesan múltiples clientes de manera secuencial.

Conclusión

A modo de conclusión, se pudo cumplir el objetivo del trabajo práctico de desarrollar un sistema que cumpliera los requerimientos pedidos: procesamiento de datasets secuenciales, alta disponibilidad y tolerancia a fallos. Para ello se tuvo que agregar una interfaz que se comunicara con el cliente para informar si es posible procesar una request en el caso en el que el sistema estuviera libre y que no es posible en caso de que una query esté siendo procesada. Adicionalmente se tuvieron que agregar distintos nodos monitores que reciben heartbeats de los nodos y así poder detectar si alguno sufría una caída a fin de poder levantarlo y garantizar alta disponibilidad. Finalmente, como en caso de una caída de un nodo se comprometía el resultado de la query, se tuvo que persistir estado en un archivo para los nodos que lo requirieran.

A partir del desarrollo del trabajo práctico nos dimos cuenta que para cumplir con los requerimientos de alta disponibilidad y tolerancia a fallos era necesario agregar un overhead de procesamiento considerablemente alto.