TP Middleware y Coordinacion - Bike Rides Analyzer

Scope

Se solicita un sistema distribuido que analice los registros de viajes realizados con bicicletas de la red pública provista por grandes ciudades.

Se requiere obtener:

1. La duración promedio de viajes que iniciaron en días con precipitaciones >30mm.
2. Los nombres de estaciones que al menos duplicaron la cantidad de viajes iniciados en ellas entre 2016 y el 2017.
3. Los nombres de estaciones de Montreal para la que el promedio de los ciclistas recorren más de 6km en llegar a ellas.

Dicha información se debe obtener de registros de clima, estaciones de bicicleta y viajes en bicicleta para las ciudades de Montreal, Toronto y Washington.

Arquitectura

Para el sistema, se consideraron 5 unidades de desarrollo, cada una la carpeta `src/system`. Todas ellas se connectan por un middleware (RabbitMQ). Estas son:

1. **input**: se conecta a el cliente, utilizando ZeroMQ. Es el punto de entrada de los registros de clima, estaciones y viajes que el sistema debe procesar.
2. **parsers**: recibe mensajes del input, en batches csv como los que le envía el cliente. Parsea los registros de clima, estaciones y viajes, y los envía.
3. **joiners**: reciben los registros parseados y agregan información del clima y estaciones a cada viaje. También quitan alguna información irrelevante para esa pipeline. Hay 3 tipos de **joiners**:
    1. **rain_joiners**: Reciben todos los registros del clima y viajes. Agrega información de precipitaciones al viaje.
    2. **year_joiners**: Reciben los registros de las estaciones y viajes de 2016 y 2017. Agregan a cada viaje el nombre de la estación de inicio.
    3. **city_joiners**: Reciben los registros de las estaciones y viajes de una ciudad específica. Agregan a cada viaje el nombre de la estación de fin y las coordenadas de la estación de inicio y fin.
4. **aggregators**: reciben los viajes con información agregada y los unifican. Hay 3 tipos de **agregators**:
    1. **rain_aggregators**: Reciben los viajes con información de precipitaciones. Calculan la duración promedio de los viajes para cada día que llovió.
    2. **year_aggregators**: Reciben los viajes con información de la estación de inicio. Cuentan la cantidad de viajes por estación para cada año.
    3. **city_aggregators**: Reciben los viajes con información de la estación de fin y las coordenadas de la estación de inicio y fin. Calculan la distancia promedio que se recorre para llegar a cada estación.
5. **reducer**: recibe los registros unificados y los agrupan para obtener la estadística final. Hay 3 tipos de **reducer**:
    1. **rain_reducer**: Recibe los promedios de duración de viajes por día que llovió y los unifica.
    2. **year_reducer**: Recibe la cantidad de viajes por estación para cada año y los unifica. Una vez unificadas todas las cantidades, encuentra las estaciones que duplicaron la cantidad de viajes de un año al otro.
    3. **city_reducer**: Recibe la distancia promedio que se recorre para llegar a cada estación y los unifica. Una vez unificadas todas las distancias promedio, encuentra las estaciones que tienen un promedio mayor a

6km.

6. **output**: sumidero de la información producida los **reducer**. Se conecta al cliente utilizando ZeroMQ y le envía las estadísticas finales cuando llegan y el cliente se lo solicita.

Objetivos y restricciones de la arquitectura

- No es necesario considerar múltiples ejecuciones del procesamiento en una misma sesión del sistema.
- No se requiere tolerancia a fallas.
- El sistema debe estar optimizado para entornos multicomputadoras.
- El sistema debe soportar el incremento de los elementos de cómputo para escalar los volúmenes de información a procesa.
- Se debe proveer *graceful quit* frente a señales `SIGTERM`.

4+1 vistas

Los diagramas de esta sección se encuentran disponibles para visualizar en app.diagrams.net. El archivo `.xml` utilizado se encuentra en este repositorio.

Escenarios

Los casos de uso del sistema son bastante sencillos, y estan muy ligados al Scope descripto en la sección anterior. Se pueden ver en el siguiente diagrama:

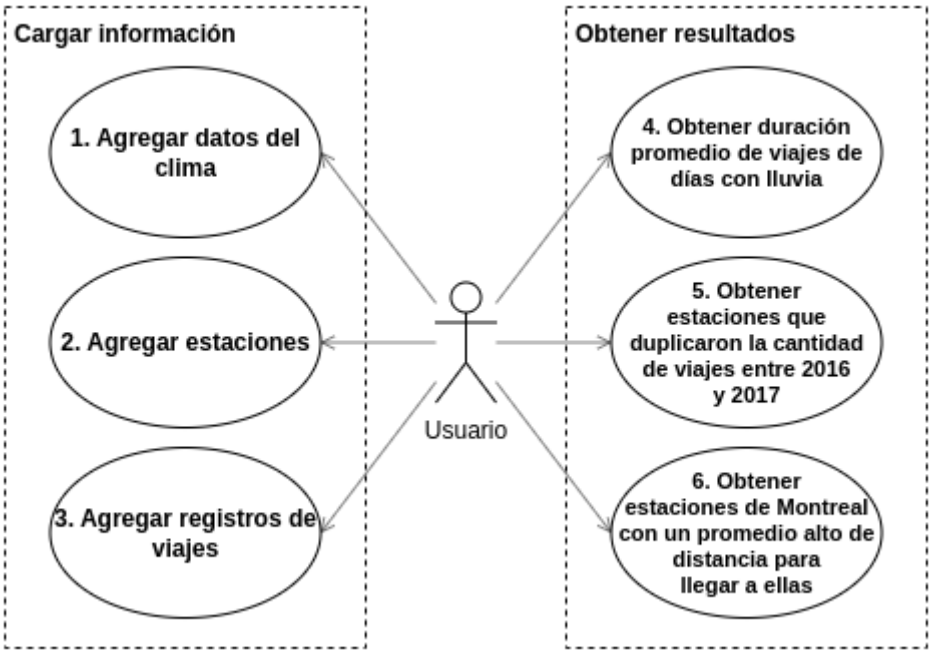


Diagrama de Casos de uso

Vista lógica

Para obtener las 3 estadísticas, hay 3 flujos de procesamiento información. Cada uno de ellas se puede ver en el siguiente diagrama:

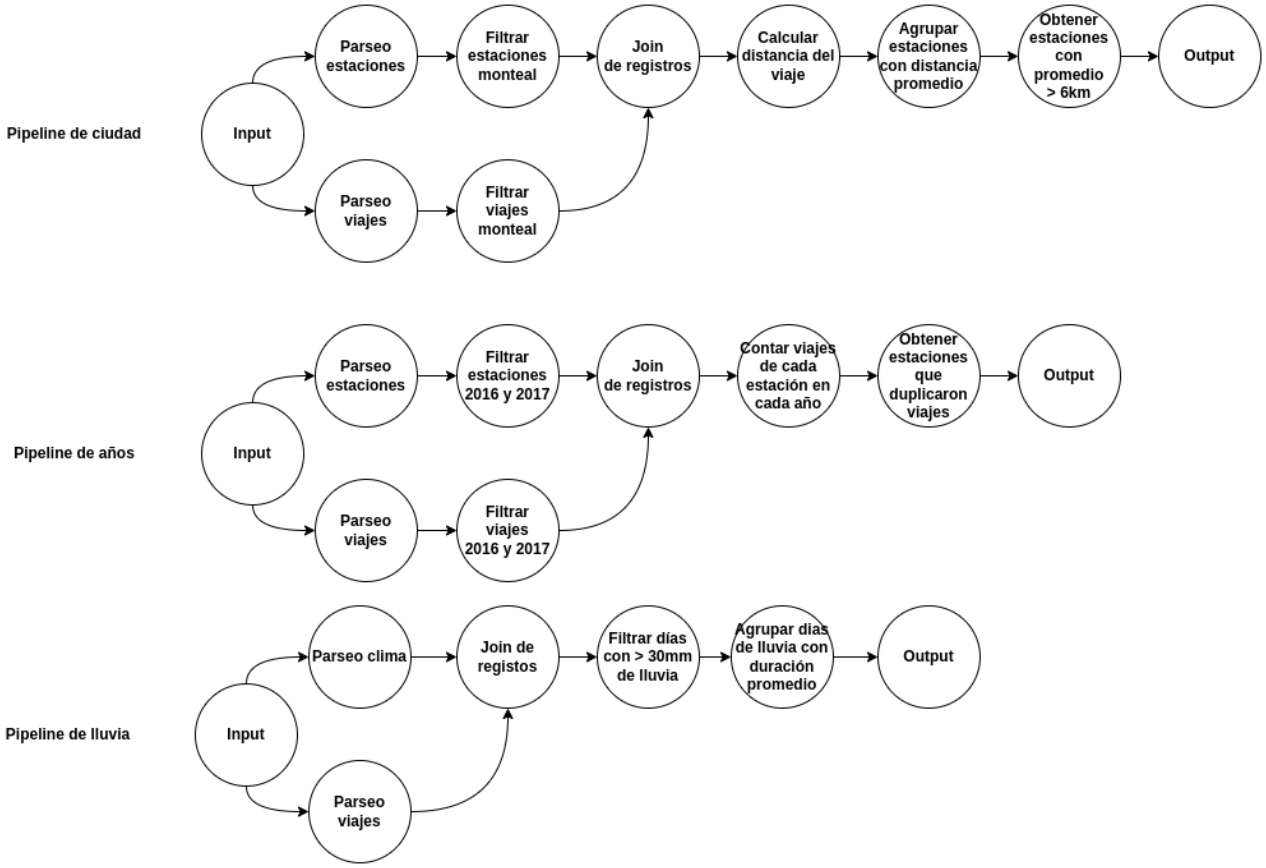


Diagrama del DAG

Vista de procesos

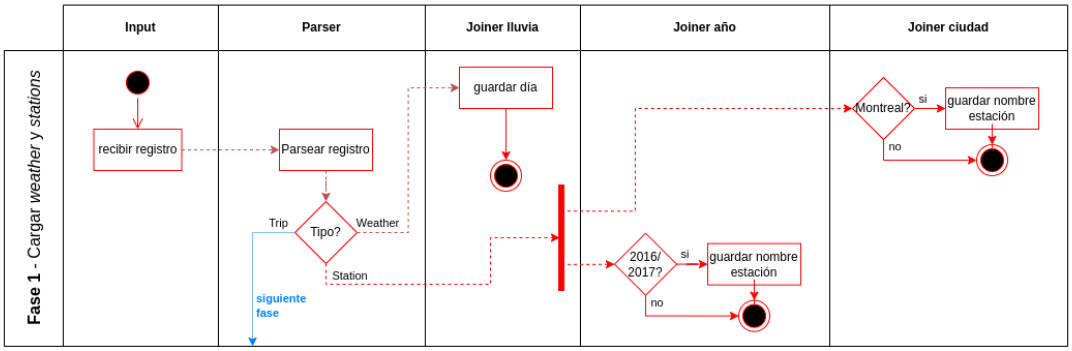
El funcionamiento del sistema, a nivel general, tiene 3 fases:

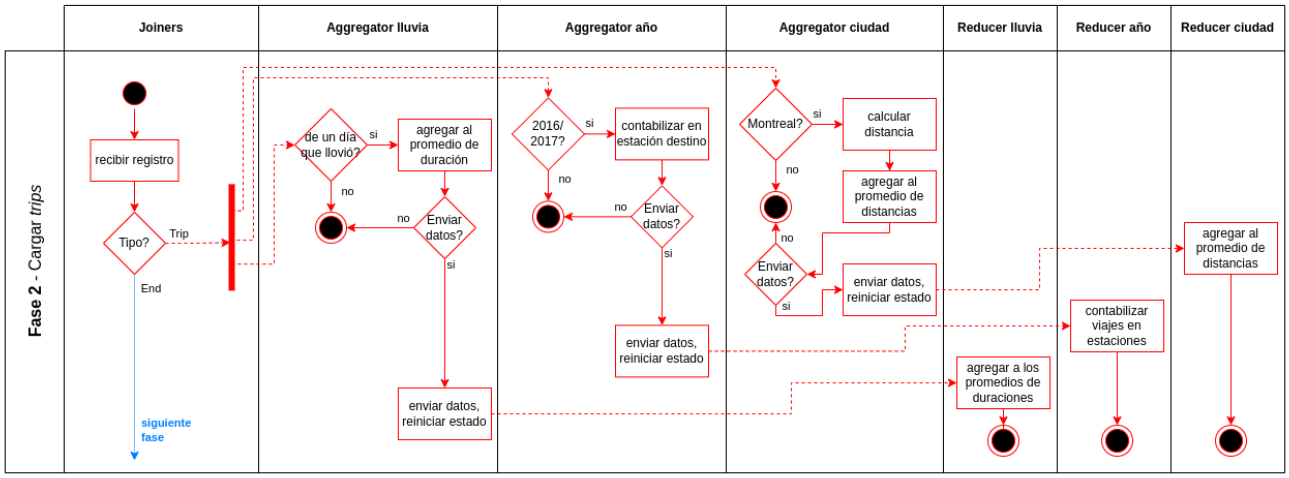
1. **Envío de estaciones y clima:** En esta etapa se envían los registros de estaciones y clima al sistema. Los **parsers** los parsean y los envían a los **joiners**, para que los almacenen.
2. **Envío de viajes:** En esta etapa se envían los registros de viajes al sistema. Los **parsers** los parsean y los envían a los **joiners** para que utilicen la información almacenada y los enriquezcan antes de continuar con el procesamiento.
3. **Finalización**: Una vez enviados todos los viajes, se obtienen las estadísticas finales.

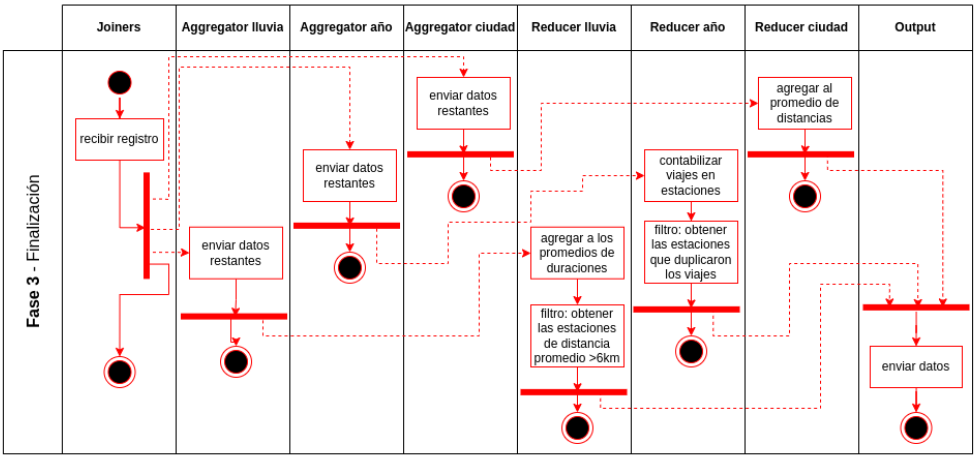
Estas fases se pueden ver en los siguientes diagramas de actividades:

> Nota: Por simplicidad, se omitió al **input** y los **parsers** en el segundo y tercer diagrama, pero siguen realizando procesamiento, solo que únicamente con viajes y no con estaciones y clima.

Actividad de un registro (por fase)



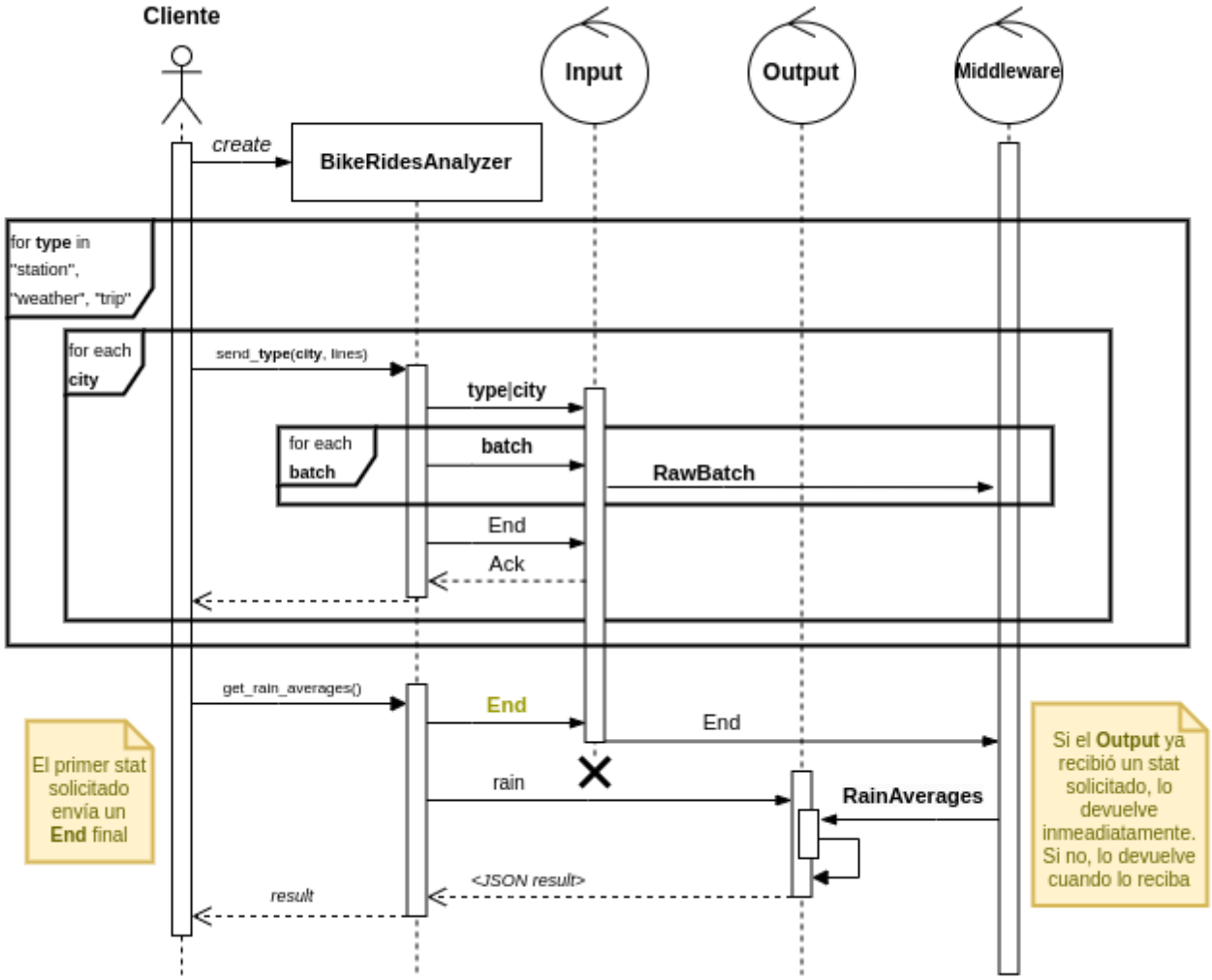




Diagramas de Actividades

Para la comunicación con el cliente, se implementó un pequeño protocolo que le permite enviar registros y solicitar los resultados. Hay un módulo $BikeRidesAnalyzer$ que actúa como interfaz del sistema para el cliente. El protocolo se puede ver en el siguiente diagrama de secuencia:

> Nota: en este ejemplo el cliente solo solicita un resultado, pero puede solicitar los 3 cuantas veces quiera.



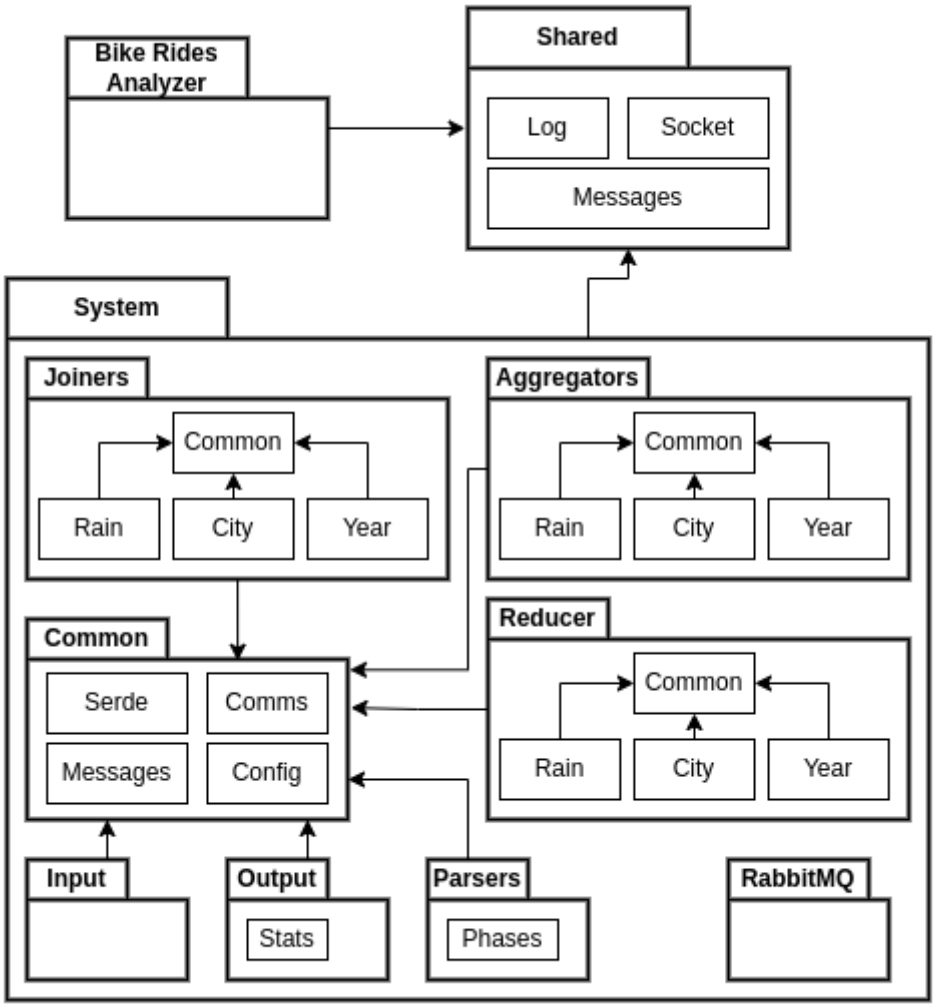
Diagramas de Secuencia

Vista de desarrollo

El código del proyecto esta separado en 3 paquetes:

- **BikeRidesAnalyzer**: La librería del cliente que se conecta al sistema y le solicita los resultados.
- **System**: Contiene los paquetes de cada uno de los nodos del sistema. También hay un paquete `common` que contiene código compartido para la configuración, comunicaciones, definición de mensajes y serialización/deserialización.
- **Shared**: Paquet de código compartido entre el sistema y la librearía del cliente. Contiene las definiciones de algunas constantes del protocolo, un wrapper sobre el socket de ZeroMQ y un configurador de logs.

Las dependencias entre los paquetes se pueden ver en el siguiente diagrama de paquetes:



Diagramas de Paquetes

Vista Física

Cada servicio del sistema se encuentra contenerizado en un contenedor de Docker. Para la comunicación entre los contenedores se utiliza otro container con RabbitMQ como middleware. Salvo el **Input** y **Output** que también se comunican con el cliente, los servicios solo se comunican a través del middleware. El diagrama de despliegue se puede ver a continuación:

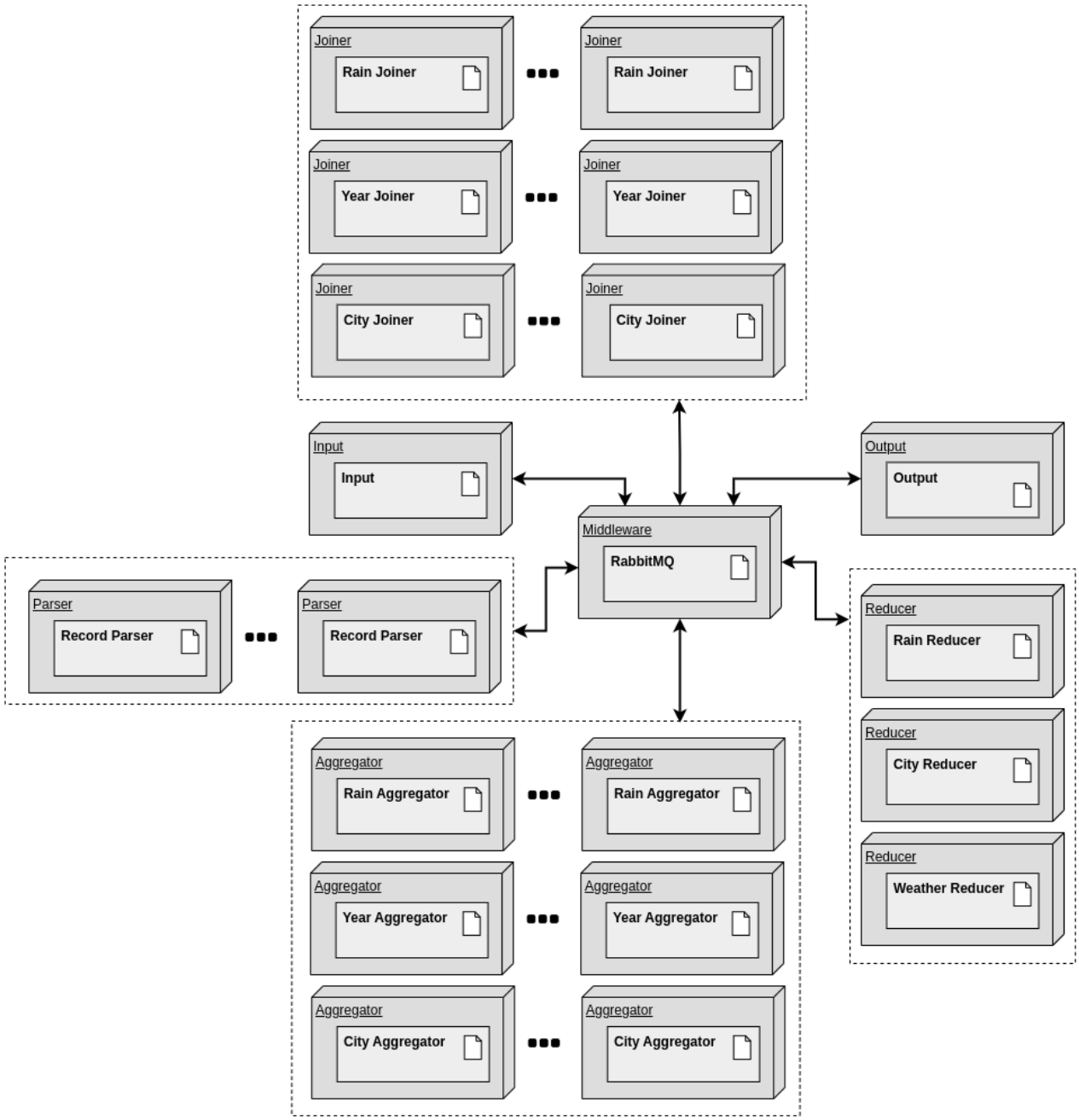


Diagrama de Despliegue

En el siguiente diagrama se puede ver como es el flujo de información entre los servicios y su escalabilidad. Los **parsers**, **joiners** y **aggregators** pueden escalarse a una cantidad arbitraria de instancias, lo cual permitiría procesar tantos registros como sea necesario. Los **aggregators** envían resultados parciales a los **reducer** cada un intervalo configurable de tiempo, el cual se puede aumentar si no se procesan lo suficientemente rápido.

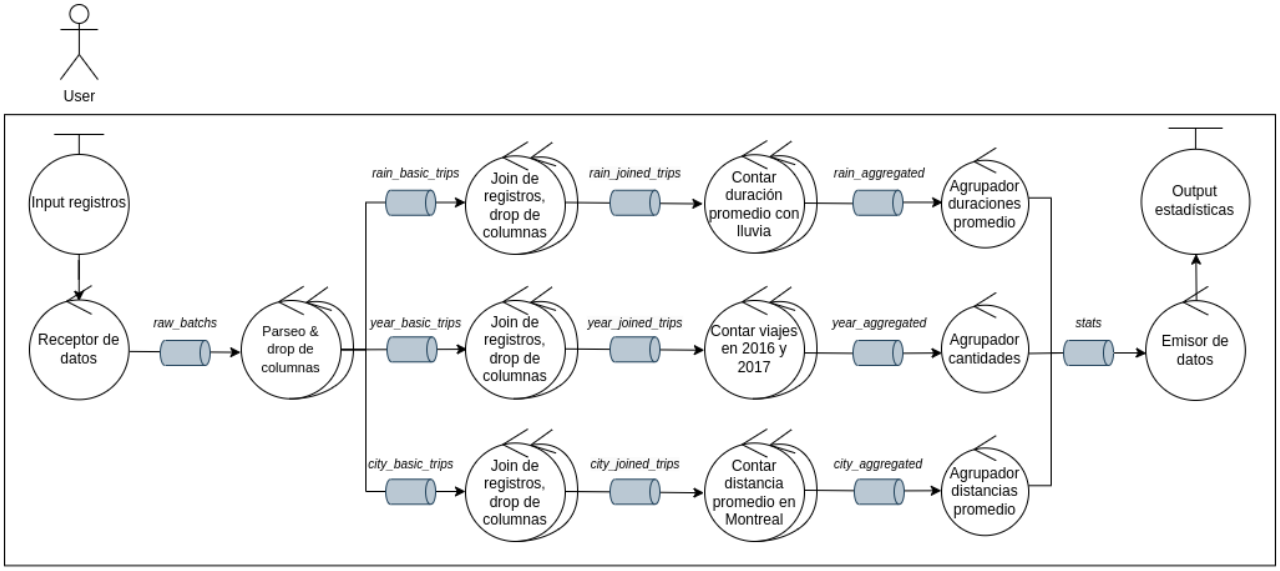


Diagrama de Robustez

Para más detalles sobre las queues utilizadas en el sistema, recomiendo ver el diagrama *Queues & Exchanges* en app.diagrams.net. Este diagrama es similar al de robustez pero va en más detalle con los exchanges y queues de RabbitMQ utilizados, los tópicos y tipo de mensajes de cada queue. En ese diagrama, se ejemplificó escalando los nodos de procesamiento a 3 instancias.