Distributed Systems I

Work Assigment 1
Bike Rides Analyzer

Facultad de Ingeniería, Universidad de Buenos Aires

Andrés Zambrano - 105500

1. Scope

The objective of this assignment is to develop a distributed system that

analyzes bicycle trips done using the public network of the following cities: Montreal, Toronto and Washington.

More specifically, the following queries need to be answered:

- Average duration of trips that started in days with precipitation higher than 30 mm.
- Names of the stations whose starting trips doubled from 2016 to 2017.
- Names of Montreal's stations to which cyclists need to travel on average more than 6 km.

The system needs to follow a Client-Server application in which the client sends the data to the server and also asks for the query results every given time.

The server must be optimized for multicomputing deploys, and needs to be able to have its computing elements increased in order to process higher volumes of data.

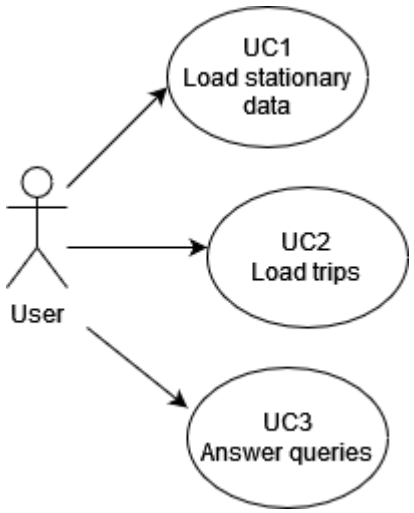
2. Software Architecture

Client and server are connected via TCP sockets. The server's inner processes are connected via RabbitMQ, while client's inner processes are connected via multiprocessing queues.

3. Architectural Goals & Constraints

Server needs to be able to have its number of computing elements modified.

4. Scenarios



The system is divided in two distinct phases. The first one corresponds to the load of stationary data (weather and stations data), while the second one focuses on processing the dynamic data (trips) and answering the queries.

Therefore, these are the three main scenarios:

1. Client sends the stationary data to the server and the latter saves it in order to answer the queries later on.
2. Client sends the trips to the server.
3. Client asks for the server to answer the three queries.

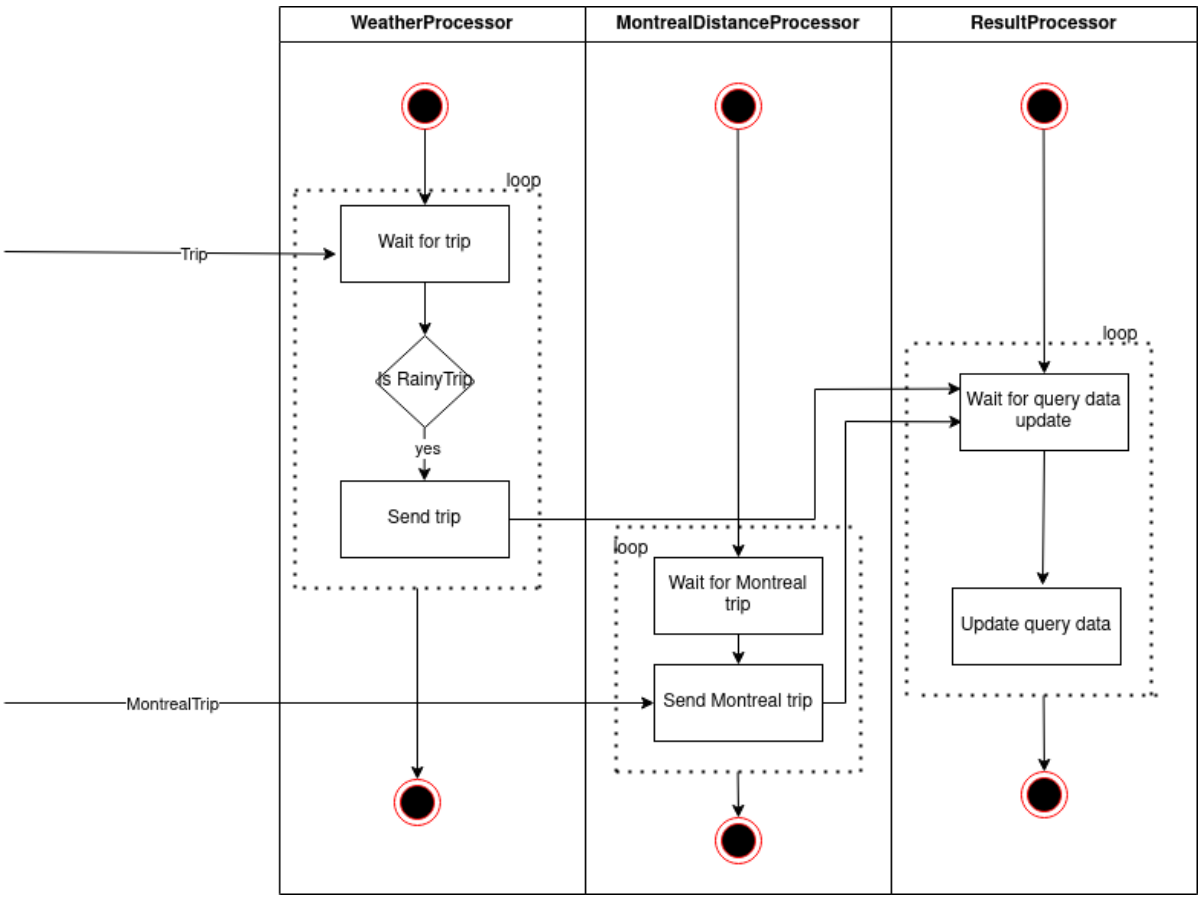
Note that the first phase happens at the beginning of the program and it is the shortest one of all, but it can happen while trips are also being sent. After all stationary data is sent, scenarios 2 and 3 run in parallel until the client finishes sending all the data and a final query is processed.

5. Logical View

No diagrams for the Logical View were made.

6. Process View

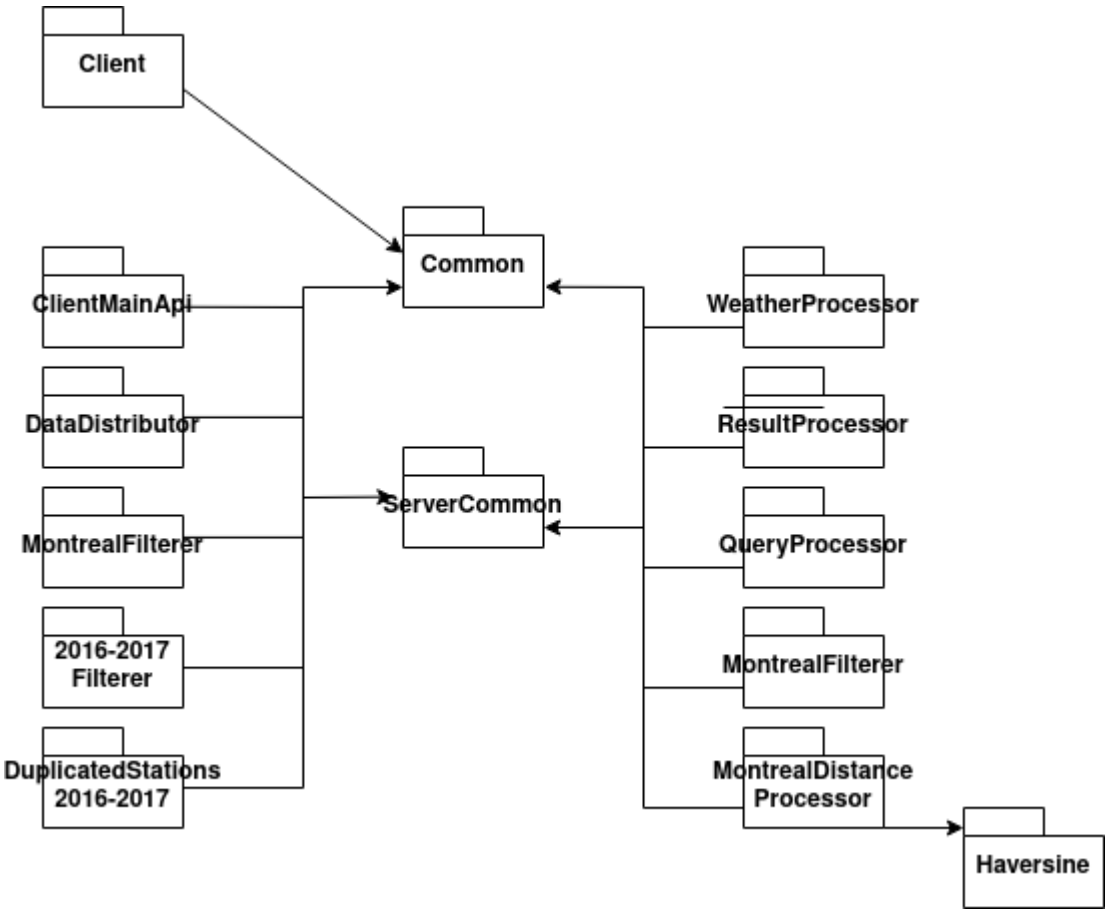
The following activity diagram shows how two different nodes interact with the ResultProcessor node.

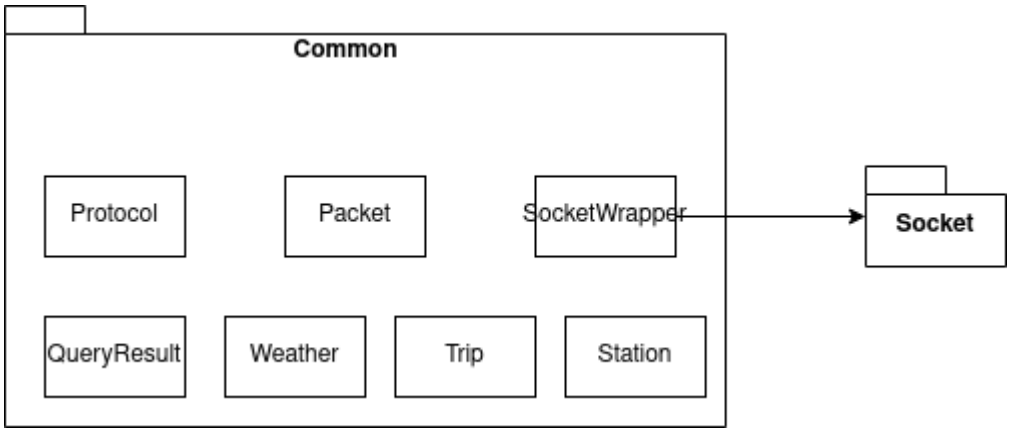


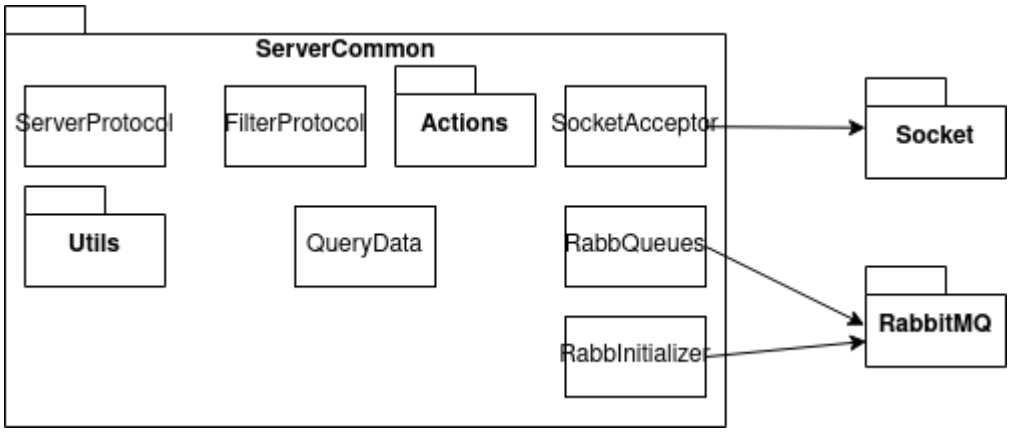
No sequence diagram were made.

7. Development View

The following packet diagrams apply:


Client
Common
ClientMainApi
WeatherProcessor
DataDistributor
ResultProcessor
ServerCommon
MontrealFilterer
QueryProcessor
2016-2017
Filterer
MontrealFilterer
DuplicatedStations
2016-2017
MontrealDistance
Processor
Haversine

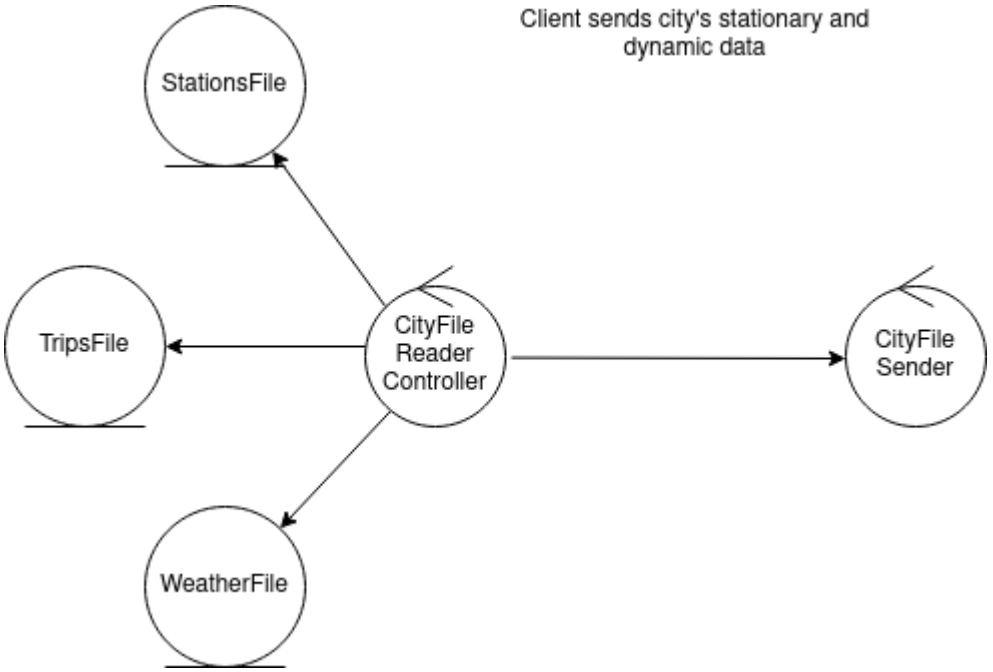

Common
Protocol
Packet
SocketWrapper
Socket
QueryResult
Weather
Trip
Station


ServerCommon
ServerProtocol
FilterProtocol
Actions
SocketAcceptor
Socket
Utils
QueryData
RabbQueues
RabbitMQ
RabbInitializer

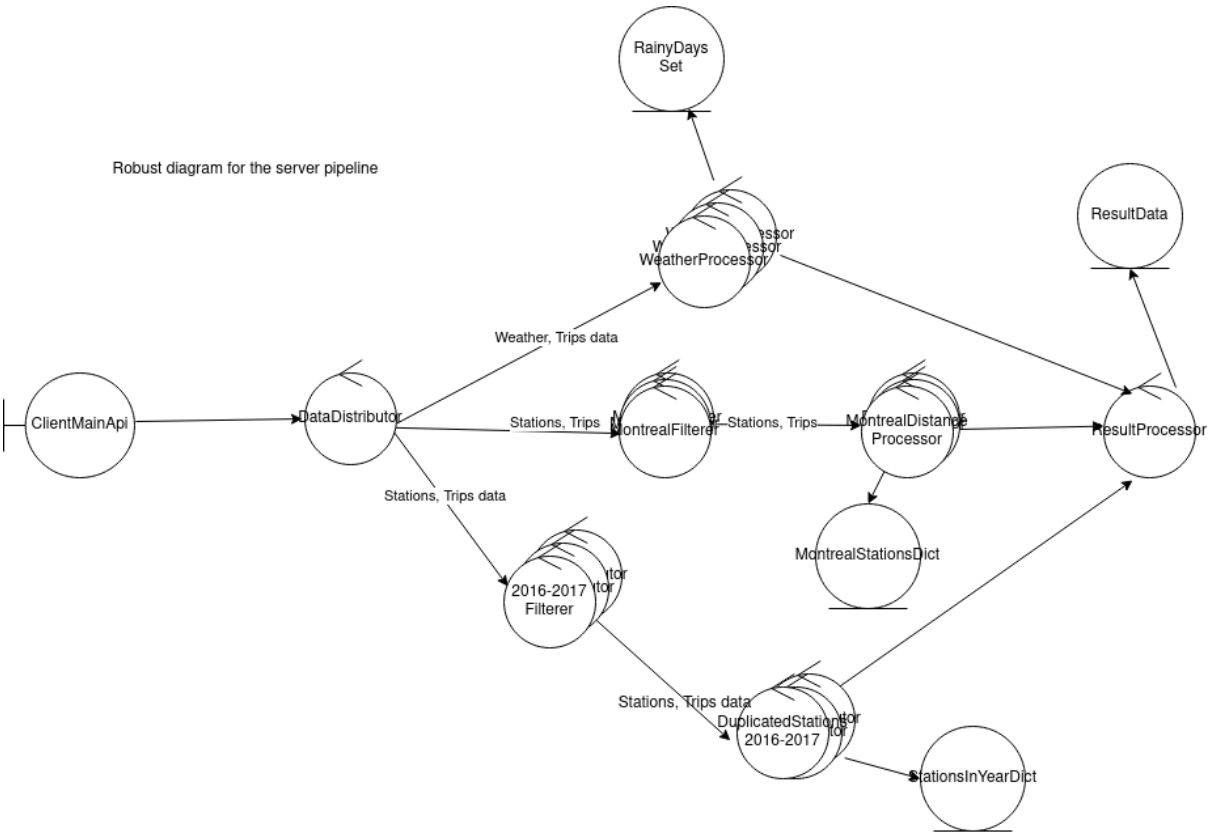
8. Physical View

8.1 Robust diagrams

For each entity in the following Client Robust diagram, there are three files, one for each city (Montreal, Washington and Toronto).



Meanwhile, the server receives the data:



When the respectives nodes receive the weather and stations data, it creates the following data structures that are later going to be used to update the queries' answers.

- RainyDaysSet: it is a set of tuples (city, day). If a tuple exists in this set, it means that in that city it rained that day. It is used by the WeatherProcessor nodes.
- StationsInYearDict: it is a dictionary of tuples (year, station_id) to tuple (station name, latitude, longitude). Used by the DuplicatedStations nodes.
- MontrealStationsDict: it is a dictionary of tuples (year, station_id) to tuple (station name, latitude, longitude). Used by the MontrealDistanceProcessor nodes.

According to these data structures, the processor nodes send trips to the ResultProcessor node, which makes the computing work to update the data needed to obtain the results for the queries.

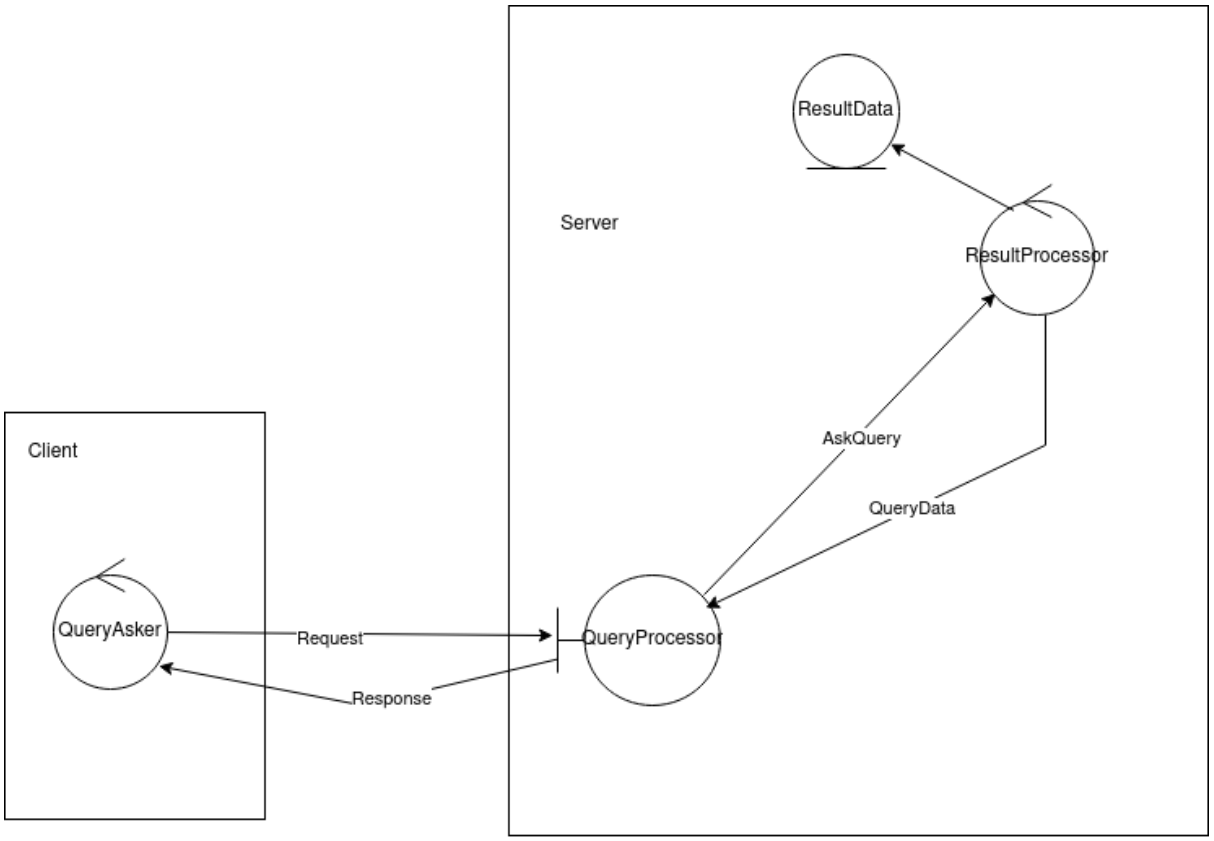
The ResultData entity is made out of the following data structures:

- DateToDurationAvg, a dictionary of date to RunningAverage.

- YearToStationToCounter, a dictionary of year to an inner dictionary of tuples (station name, city name) to counter.
- StationToDistanceAvg, a dictionary of station name to RunningAverage.

8.1.3 Answering queries scenario

In a previous diagram, a process on the client side was shown, which is in charge of periodically asking the server for the query result. At the same time, the server expects to answer the query.



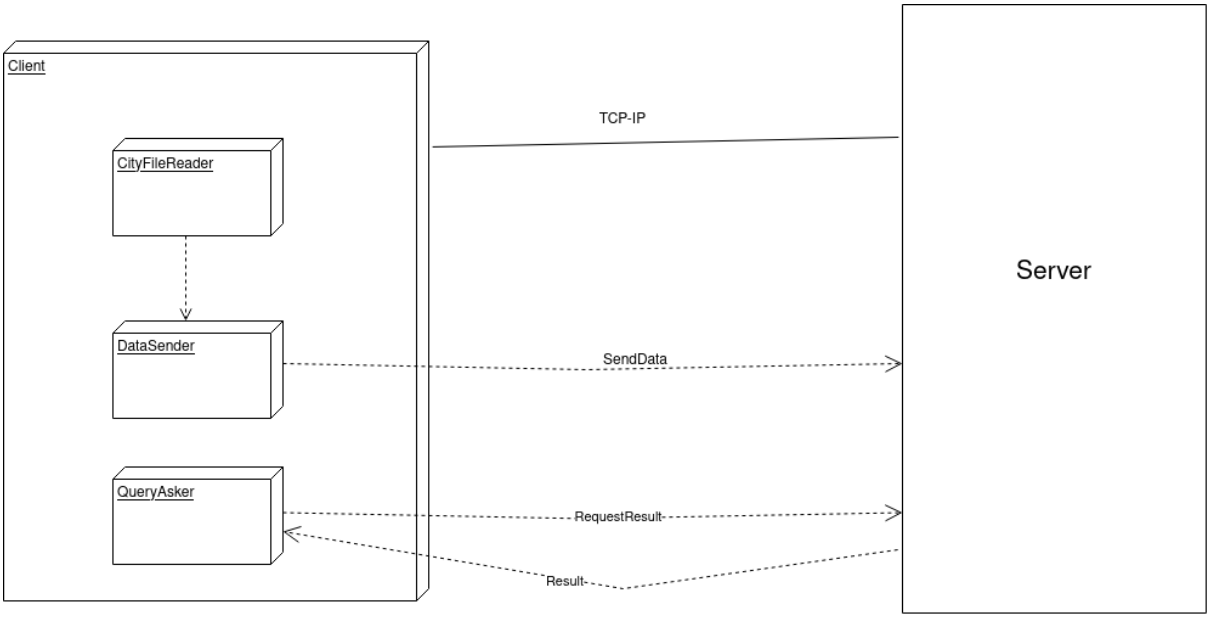
The QueryProcessor receives the same data the ResultProcessor has in its ResultData, and it uses it to calculate and create the queries results, which is made out of structures one would expect given the queries:

- RainyDateNAvgList: list of tuples (date, avg). Ordered by date.
- YearToStationToCounterList: list of tuples (station name, 2017 count, 2016 count). Note that only the stations that duplicated its trips are in this list.
- FarAwayStationList: list of tuples (station, distance_avg). Note that only the stations whose distance average is greater or equal than 6 are shown.

This data is sent to the client who prints it on the screen. If it is the final result, it saves the data on a file.

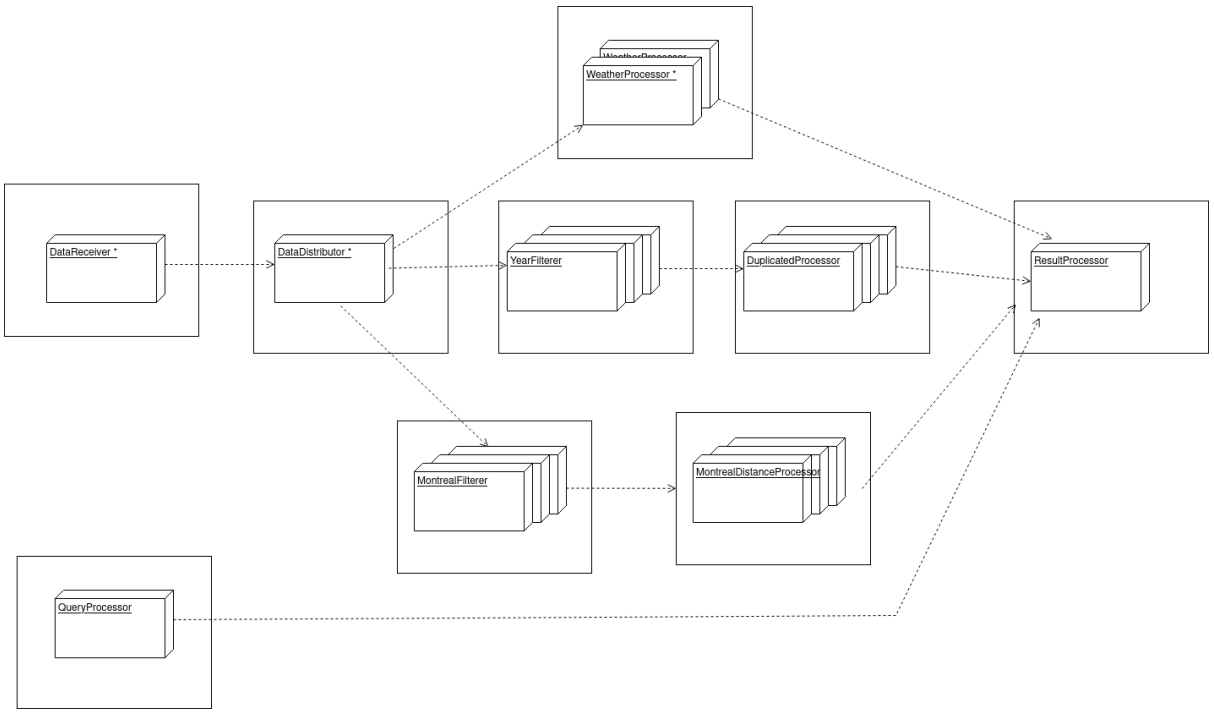
There is an important observation to make: it is a request-response message, so the client will not ask for another query result if the server has not returned the result of a previous query.

8.2 Deployment Diagram:
For the client the following deployment diagram applies:



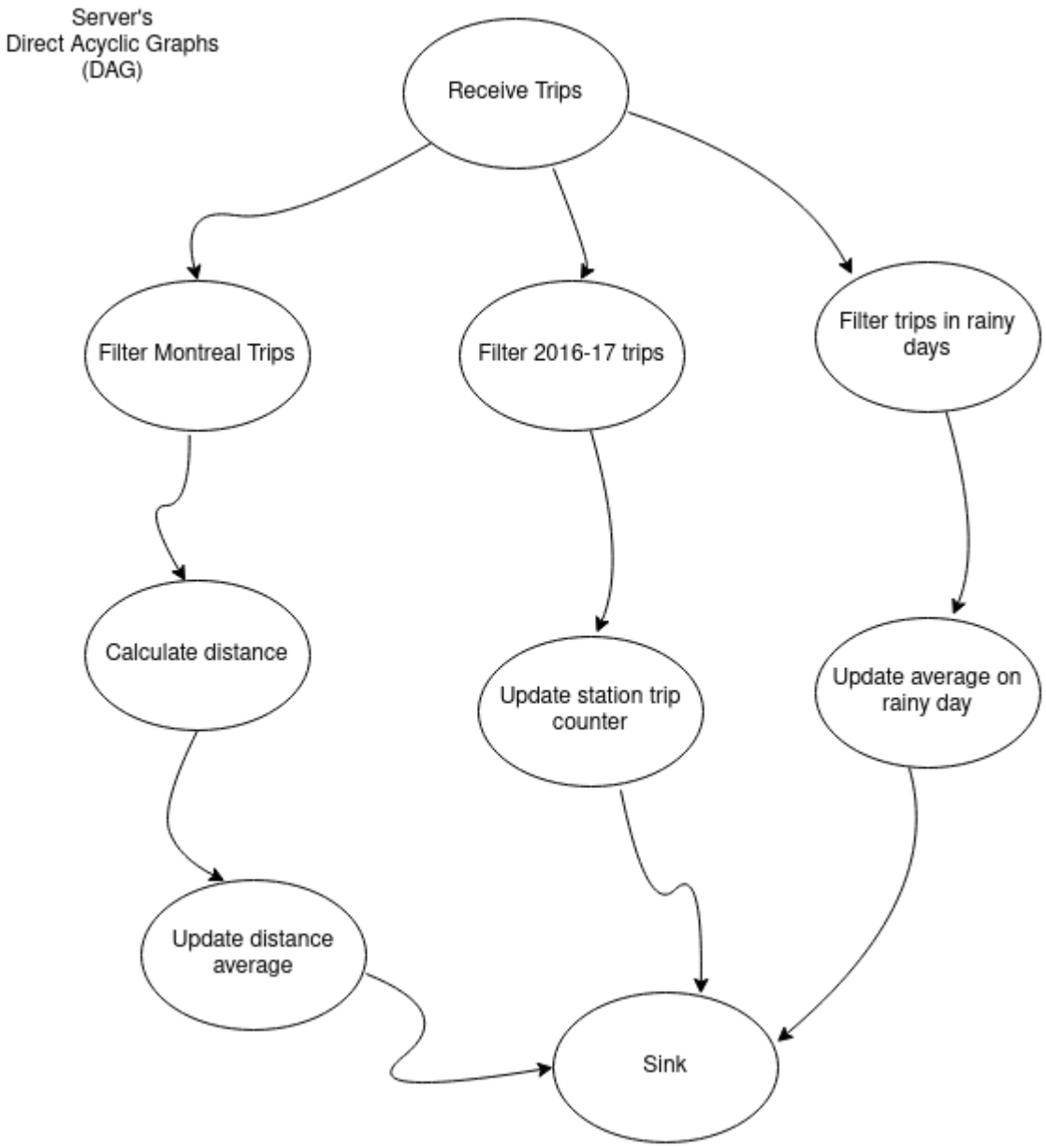
All processes are in the same machine, having access to the data files.

Meanwhile, for the server the following deployment diagram applies:



Note that the server's processes do not have to be all in the same machine.

9. Direct Acyclic Graph


Server's
Direct Acyclic Graphs
(DAG)
Receive Trips
Filter Montreal Trips
Filter 2016-17 trips
Filter trips in rainy days
Calculate distance
Update station trip counter
Update average on rainy day
Update distance average
Sink