Fog Computing - Prototype

Klinkert Daniel , Demollari Elio

July 11, 2022

Contents

1 Introduction **2**

2 Implementation **2**

2.1 Patterns used 3

2.2 Improvements 3

1 Introduction

Public Transport Tracking System is a prototype that simulates the interaction between two virtual sensors and a cloud component.
Our idea is to simulate a tracking system for public transportation. The local component represents a sensor placed in a bus station and constantly sends data to the cloud, containing information such as the current bus arriving at the station, the average speed of the bus, and the route that this bus covers. On the other side, we have the server processing the data from different sensors and sending back the information so that every sensor can display in live time the bus arrival; both applications where developed using python.

2 Implementation

The system uses a high-performance asynchronous messaging library called ZeroMQ, which supports messaging patterns such as pub/sub, and request/replied used in our project to build inter-processing messaging between the local component and the cloud. The first objective about this prototype was to be able to reliable message delivery, therefore we needed a design that allows us to keep track what was sent or not, which means if something goes wrong, if the client present network issues, the components must keep working and queuing the data to deliver once the re-connection is established. The Lazy Pirate Pattern [1] waits until the server acknowledges the request from the client to continue sending other request. One important benefit of using ZMQ is that the request are queue and get dispatched using FIFO, therefore when the server is not reachable the client will keep trying to send the request until the server replies back. This is an advantage because the client does not need to keep track of the request send and is guarantee that the will be send in a specific order, but the disadvantage comes when the queue is blocked because the server has not replied yet. In our project the disadvantage was not critical because if the server was not responding a new request will also fail, However, we saw that the thread was block waiting, thus the data was not generated anymore, that was the main reason to use threads, because we were using threads we needed to keep the variables shared between them save, therefore we took advantage of the data type Queue [2] defined in python that by default has a thread save approach. On the client, we have a script that constantly generates random data (to simulate the bus arrivals), this data is stored in the queue, and once the client connects with the server, it picks data from the queue sends it.

[1] https://zguide.zeromq.org/docs/chapter4/#Client-Side-Reliability-Lazy-Pirate-Pattern
[2] https://docs.python.org/3/library/queue.html

2.1 Patterns used

In our prototype we decided to use multiple patterns to try to fulfill the requirement of a reliable messaging application, the first pattern we used was the Request-reply, it was used from the client to the server to make sure the server was available to send further information. The request was send from the client and like mentioned before we waited until the server replied back to send more information using a Pub-sub where the server also a subscriber for the station, this decision of making the server a subscriber was made to avoid the server to know the ip address of each client therefor making the app in the server side more scalable. Additionally the server side also has a publisher socket that publishes information to the registered stations, the stations register to the server using the Request-Reply pattern mentioned above. During the development of the prototype we faced different issues, the most impact-full was the decision to first develop the applications using NodeJS as the team felt more comfortable with it, however we ended up using python because of the structure examples and documentation of the library ZMQ.

2.2 Improvements

Using the ZQM library has a lot of advantages, it support quite complex messaging patterns that we could implement to make this prototype better, the first change we would suggest is to instead of the Request-Reply patterns use the Request-Router messaging patterns, where the server applies the router socket, thus it is able to process multiple request at the same time, which is the restriction of the Request-Reply, we added a simulated latency of 2 seconds every time the server gets a request from the client, if within those two seconds we would have another client requesting the server for a reply, the server first will need to wait those two seconds to process the second client request, which is not optimal but for a prototype is more than enough. On the other hand, a second improvement could be to write the data to disk to make sure if the server or the client crashes we still have the data, in this prototype we cover only the part where the connection was not reliable therefore the in memory variables were enough.