Performance and Fault Tolerance

Plan and Go:

Visual Public Transportation Analysis Tool

Quality of Service (QoS)

Topic: external

Publisher QoS: **0**
Component: **requestGenerator**
We are sending a lot of messages, which includes random coordinates from here, however, it will not hinder the City Planner from performing analysis as there is a stream of requests coming through. As there are a lot of requests, the messages will still form a trend even though some messages drop or are sent multiple times.

Subscriber QoS: **0**
Component: **requestPipeAndFilter**
There will be a lot of messages being received, therefore, it is not optimal to ensure that all are received. It will be fine if some messages are lost or duplicated and this will not affect the City Planer's analysis.

Topic: requests

Publisher QoS: **1**
Component: **requestsPipeAndFilter**
The number of messages will be sent from this component will be fewer compared to topic "external" because this component implements a queueing system to manage the API requests, therefore this limits the rate of requests in Västra Götaland. This means that messages will be published at a reasonable rate, therefore, it is justifiable to ensure the broker sends all of these messages. Here, we want to ensure those requests to be sent at least once, which makes sure there will be enough data for City Planners to perform analysis.

Subscriber QoS: **2**
Component: **visualizer**
We want to make sure that every request located in Västra Götaland is received, since the amount of those requests might be few, since we are receiving them from a component that implements a queueing system. For correct identification of trends and bottlenecks performed by the City Planners we decided to use a higher level of QoS to ensure that messages will be received exactly once.

Fault Tolerance Mechanisms

To ensure our system is fault tolerant, we have implemented a queue in the RequestsPipeAndFilter component.

The “onLandFilter” and “inGothenburgFilter” subcomponents both require external APIs. Due to budget constraints, we are limited with how many requests we can make to these APIs per minute.

The implementation of a queue as a fault tolerance method ensures that we don't exceed the limit allowed for the API calls per minute. Therefore “onLandFilter” has to send requests to the API every at least eight seconds and “inGothenburg” at least every two seconds. By implementing a queue we can avoid a ‘429- too many requests’ error.

The queue saves all messages received. As the requestsPipeAndFilter runs, it checks the queue and processes the message at the front.

Impact of Architecture

Scalability

We chose the Pipe and Filter architecture style, which will increase the scalability of the system. The filters are independent modules, which can be added or removed easily to prevent bottlenecks, so that we can easily adjust our system.

Throughput

Setting the Quality of Service (QoS) at zero for certain components increases the throughput as the broker will not ensure that the message is received. However, where we have a higher QoS, this decreases the throughput, as the broker will ensure that each message is sent at least once. This increases the amount of data being transferred with every message published, which decreases the throughput rate.

Latency

The QoS is set to 0 for the ‘external’ topic. This means that the latency is consistently low for this part of the system. However, the latency of our system overall is largely controlled by the requestsPipeAndFilter’s queueing system. Each request will be processed through the pipe and filter at every seven seconds, to ensure we are within the limits of the APIs.

Availability

The queuing system we have in place limits the risk of the components crashing. Also, the components do not crash when the messages are wrongly formatted. However, we have not implemented a mechanism to ensure our entire system is always available. In the future we would implement a process that restarts the component if it should stop working.

Consistency

As we are working at component level, we do not have an issue with consistency across multiple instances of the Visualizer. However, if we were to scale up the system, the consistency might be decreased. To combat this, we would implement replication to increase the consistency of our system. For example, to implement the Pipe & Filter on two different devices instead of one and check the consistency.