# Csc8110 Report

By Huan Xu

## Overall Structure

I finished all part of this project, the structure is shown as follows:



## Smart Speed Camera

#### 2.1 Camera and Vehicle Information

As followed by the requirement, I choose to get Camera Information by parsing Cmd line arguments. The required cmd arguments include ID, Street, City, Max speed which defined a particular Camera, in addition, to simulate vehicle passing, I also take an Integer value as vehicle passing rate.

The vehicle information includes registration plate, type of vehicle, current speed and capture time. We present two strategies to get registration plate, the first one is totally random generated, another is to random select plates from a file. The reason we chose the latter one is in part4 we need to get the over speed vehicle information along with all its history. If the plate is totally random generated, it would rare have history information. So I random select plate in a small range that makes the query in part 4 more meaningful.

The current speed of vehicle is generated by a random Gaussian function. That because in realistic scenario, most vehicles are passing with similar speed, and only a small number of vehicles are too fast or slow. And random Gaussian could simulate this.

To distinguish two different kind of message, we set a new property “Type” to profile or message in brokered message, and consumer could use Type property to judge which kind of message it is.

#### Send Message and offline resend

I use two thread to send message and trying to resend message at the same time. The design is shown as follows:



The LinkedBlockingQueue will block when there is no data in it, I use this feature to control this two thread and automatic resending message when it failed.

#### Message coding

The borkeredmessage accept String or Input steam as body. To minimize message size, we serialize class to json rather than direct serialize it as objects.

## NoSQL Consumer

#### 3.1 AMQP VS HTTP(RESTful)

Based

on the requirement, we need to reduce the number of requests and increase the performance. As suggested by [Optimizing performance](https://docs.microsoft.com/en-gb/azure/service-bus-messaging/service-bus-performance-improvements), Service Bus support three different protocols: AMQP, SBMP and HTTP. SBMP is only supported by .net, so I do not consider this choice. HTTP is supported by the Azure Api by connecting with Restful service, each time it starts communication, it will construct a new TCP connection to Service Bus. AMQP, on the other hand, directly based on TCP and it will maintain the connection as long as the message factory exists. That makes the AMQP more efficient than HTTP. In addition, AMQP support batching and prefetching, which caches and receives a group of message in one request. This feature will reduce the number of requests. However, the drawback of AMQP is it not support topic or subscription creation. As a result, in NoSQL consumer, I use HTTP based Azure API to create and check Topic and subscription, then using AMQP based Apache Qpid + JMS to receive message, the design could be found in Figure 1.

In implementation, I set prefetch policy in configuration file (jndi.properties) to enable prefetch and batch receive, and use asynchronous message listener to handle batched message that makes it work the same as single receiver.

#### Camera Entity Design

The table storage use Partition Key and Row Key to unique identify an entity. Partition Key is also used for distributing entity to different partitions. And the Table storage is scaled based on message in one Partition, which means to optimize the preference for searching and scaling, we need to balance the number of partition key and row keys in one partition.

The documents do not describe the details of camera usage and coverage. So in different scenario, we need to choose different partition key and row key.

If the information of camera is usually changed, we choose id as partition key and star time as row key.

if the camera only located in limited cities and are most in fixed location, we may choose street as partition key and combine id with start time as row key.

If the camera cover a large number of cites, we may choose city as partition key and combine id, start time as row key.

In the implement, I just choose the first one as an example.

#### Vehicle Entity Design

The Vehicle Entity includes the details of camera information, the reason of it is Nosql database do not support database level join operation, so we usually use more spaces for faster speed. That's why I duplicate the camera information.

To unique identify each record, we choose captured camera id + start time as partition key, and registration plates + captured time as row key. We need to add camera information in key is because each vehicle could be captured by multiple cameras at same time. And we include reg and time in row key at the same time, because one camera could capture same vehicles multiple times.

#### Batched table operation

Camera information is limited, so I just insert one entity each time. For vehicle information, with the increasing number of camera, the number of entity could be very large. So I use batched operation.

Batched operation for table storage required each batch only contains same partition key. So in implementation, I choose to use Map<PartitionKey,Entity> as cache for batched insert. The number of batch size could be set in Config class.

## Police Monitor

#### 4.1 Subscription Rule

The rules in subscription only support filter Properties in BrokersMessage. So when smart camera sending vehicle message, it will set Overspeed property as max speed – current speed. In police monitor, we add rule only accept message with overspeed > 0.

#### 4.2 Over Speed Vehicle Entity

The Over Speed Vehicle Entity is similar with Vehicle Entity, it only contains two more columns, the number of overspeed and whether it is PRIORITY.

## Query Application

#### 5.1 List Camera Information

This's easy one, I just use azure api to list all camera.

#### List Over Speed Vehicle and All its History

Table storage do not support database level join, so I use application level join. I firstly get all entities with Priority = true in Over Speed Table and then use each registration plate to search its history in Vehicle Table.

## Vehicle Check

Vehicle check simulate long time process, which makes single program can’t fit for requirement. To solve this problem, I create multiple classic VM in same set and scaled (turn on and shut down) by Queue length.

I assume the long time process will finish in 5 seconds. So it can handle 12 entities per mins.

In setting, I set the scale wait time to 10 minutes and the target per machine is set to 120.

In each classic VM, I use upstart to automatic start vehicle check program after the file system is ready.