



Responsible Prof.

Jérôme Härri

Semester Project

An IoT framework for Traffic Jam detection in Smart Cities

SPARTALIS Alexandros Ilias
Romain Velu
Shule Li

Abstract

The following project is part of an IoT framework for Traffic Jam detection in Smart Cities designed to detect and handle possible traffic jams using real time vehicular data. The framework is divided in three parts, -vehicular data generation using SUMO, development of techniques to detect traffic jams, and study of CPM message- and every each of them complements the needs of the project while keeping also the ability to operate in an autonomous way .

The first part, which will be explained vastly in this report constitutes the primer of the project. In lack of real vehicular data, it is considered appropriate the use of the virtual simulation platform SUMO, in order to generate all the necessary vehicular information that are indented to be exploited for the needs project. Furthermore it is advisable to create the project in a way that is high-scalable and easily reusable for various scenarios.

In this report we focus on the part of the SUMO configuration explaining its capabilities and its operation. We present a model that creates an example scenario based on which we extract all the valuable vehicular metrics. We also implemented the communication between the SUMO application and the Geoserver where we update its database with the vehicular data extracted from SUMO.

Contents

Table of Contents

Abstract	2
Contents	3
Figures	4
1. Introduction.....	5
1.1 Situation	5
1.2 Problematic.....	5
1.3 Concept	5
2. Study / State of the Art.....	6
2.1 Simulation Tool SUMO	6
2.2 Traci extension	7
2.3 VANETZA stack.....	7
2.4 Geo-server	7
2.5 Network map primer	8
3. Implementation / Results.....	9
3.1 TraCi coding	9
3.1.1 Subscription method	10
3.1.2 Simple iteration method	10
3.2 JSON Output file.....	11
3.3 Connection to the Geo-server and data storage	14
4. Conclusion	16
5. Future Development.....	17
References.....	18
Appendices.....	19

Figures

Figure 1 SUMO Dlr.	6
Figure 2 Authentication	8
Figure 3 SUMO scenario network	9
Figure 4 Flows generation	9
Figure 5 Traci script - Subscriptions.....	10
Figure 6 Traci script - Custom	11
Figure 7 JSON file	12
Figure 8 JSON entry	12
Figure 9 Vehicle flow west-east	13
Figure 10 Timestamp = 1	13
Figure 11 Timestamp = 7	14
Figure 12 Communication with Geoserver.....	15
Figure 13 Database data.....	16

1. Introduction

1.1 Situation

Intelligent transportation systems (ITS) and their applications, have lately met huge hype, with a continuously increasing amount of people talking about techniques that get the road traffic management to a whole new level. Cooperative vehicular communication systems is the key to design, implement and maintain these kind of ITS.

As most of the modern vehicles are nowadays equipped with state of the art sensing technologies, we can use their collected data in our favor to have a better understanding of the surrounding area. By sharing wirelessly information between vehicles (V2V) or between vehicles and infrastructure (V2I) we can develop traffic monitoring techniques which are characterized by high accuracy and low operational cost.

Smart Cities throughout the world have been integrating sensing systems into road infrastructures to monitor important parameters such as traffic density, air pollution or any other abnormal situation in order to succeed in a better traffic management. These solutions could have a positive impact on the traffic problem of big cities, but there are two critical drawbacks that actually force us to develop better solutions. The enormous cost of a static sensing network and the limited sensing range of the above. On the contrary, cooperative vehicular systems will exploit the on-board sensors of the future connected vehicles sharing the collected data wirelessly with the infrastructure nodes, using specific reserved communication channels. That way the infrastructure will act just as a "collector", "distributor" or "forwarder" of these information drastically reducing the operational costs and opening new horizons in the capabilities of such a network.

1.2 Problematic

As mentioned before, various data extracted from the vehicles are the main prerequisite in order to build all the promising techniques that will regulate the mobility in a smarter and more efficient manner.

Unfortunately, all the new technologies (sensors, communication protocols, smart infrastructures nodes, etc.) are still under development or testing mode which leads us to the most important obstacle, the lack of real data. It is easily understood that any research based on real data, gathered from the actual running vehicles in the road network would of course lead to more realistic solutions and results. Nevertheless, the use of dedicated software can fill the gap and help us progress in our research.

More precisely all the essential information is going to be generated using the mobility simulator SUMO and a Geo-server will act as the road infrastructure that is in charge of collecting and distributing raw or processed data. Our goal is to build a framework that is going to operate similarly to the real scenario, emulating vehicles and infrastructure. When we get to the point where our implementation works as intended to be, we can adapt it to the real conditions and re-test it.

1.3 Concept

The project is part of an IoT framework that is being developed in EURECOM'S laboratories and is able to detect Traffic Jam in Smart Cities.

The main idea behind that, aiming to be tested on real-world situations, is to build a tool operating on the cloud, that could exploit the real-time vehicular data in order to detect situations with high traffic jam potential and take decisions on the fly to regulate the mobility accordingly.

As most of the time mobility is unpredictable and fluid there is no perfect solution when it comes to determine the most efficient vehicle route for a specific environment. So far we were able to study the mobility behavior but only at a distinct later date, which could let us have an overview of the average mobility scheme but couldn't have access instantly to the actual situation.

With the new technologies that are incorporated more and more in the new generation vehicles, we are closer than ever to achieve that. Taking advantage of the multiple sensors that cars are equipped with, as well as using all the late V2X communication protocols we are able to track vehicles easily and use their sensing information to build a more sustainable mobility environment.

Some cases that would be useful to have a functional framework like this are :

- Peak time traffic in specific areas where it would be more advisable for a driver to choose another longer route in terms of distance, but way faster because of an ongoing traffic jam .
- Car accident situations where a part of the road is blocked.
- Emergency vehicle in need of empty lane to pass through the traffic.
- Adaptive side road signs to regulate the maximum-minimum speed of the vehicles according to the traffic density

2. Study / State of the Art

2.1 Simulation Tool SUMO



Figure 1 SUMO Dlr.

For the needs of the project we used SUMO environment to create the mobility scenario and TraCI extension to extract the information of the vehicles running on SUMO.

The "Simulation of Urban MObility" (Eclipse SUMO) is an open source, highly portable, microscopic and continuous road traffic simulation package designed to handle large road networks. "Eclipse SUMO" is a trademark of the Eclipse Foundation and it is mainly developed by employees of the Institute of Transportation Systems at the German Aerospace Center. *Figure 1*

TraCI uses a TCP based client/server architecture to provide access to SUMO. Thereby, SUMO acts as server that is started with additional command-line options: **--remote-port<INT>** where **<INT>** is the port SUMO will listen on for incoming connections.

When started with the **--remote-port<INT>** option, SUMO only prepares the simulation and waits for all external applications to connect and take over the control. Please note, that the **--end <TIME>** option is ignored when SUMO runs as a TraCI server, SUMO runs until the client demands a simulation end.

When using SUMO-GUI as a server, the simulation must either be started by using the play button or by setting the option **--start** before TraCI commands will be processed.

2.2 Traci extension

In order to extract the information from every single vehicle incoming to the specified area we chose we needed to run the simulation through the TraCI extension. The port in which we randomly chose for SUMO to wait for an incoming connection is 8888 and is specified under the **<traci_server>** tag of “map.sumocfg” file inside the /scenario folder.

Then, all we have to do is to initiate the TraCI in the TraCI control loop using the same port so that from now on TraCI takes the full control of the simulation. Inside this loop we can modify anything we want TraCI do for us, for example extracting position data as we will explain later.

The file that handles the TraCI extension can be implemented in either C++ or Python. For convenience we chose to use Python.

2.3 VANETZA stack

The extracted data from TraCI should be forwarded to a Geo-server through Vanetza stack using linux socket. There after some necessary data processing we could end up with some valuable results regarding the surrounding vehicular environment.

Vanetza is an open-source implementation of the ETSI C-ITS protocol suite. Though originally designed to operate on ITS-G5 channels in a Vehicular Ad Hoc Network (VANET) using IEEE 802.11p, Vanetza and its components can be combined with other communication technologies as well, e.g. GeoNetworking over IP multicast.

The script that would handle the communication between the Vanetza and TraCI wasn't available and so we decided to use HTTPS request as communication method in order to boost the progress of the project.

2.4 Geo-server

The Geo-server that is used to accommodate the receival and distribution of the various vehicle information is already set up for a previous project so we can use the already implemented API in order to subscribe and authenticate our user (TraCI sending data to the server).

Base URL for APIs – <https://geoserver.eurecom.fr/>

The above API Gateway is the entry point to the Geoserver web services. It is responsible of the authentication and for analyzing requests and route them accordingly (to other protected internal web services) based on a “**services.json**” file that relates paths to the API. This file has to be updated to accommodate the extra storage we are attempting to do.

This web service interacts with the other internal web services of Geo-server through REST requests.

The function of the API Gateway middleware is to verify if the coming request is authenticated to access services or not. It verifies the validity of the token. Before heading for the right service, the request is analysed in the middleware and the JWT is extracted from the “x-access-token” HTTP header. The middleware validation steps are shown in *Figure 2*.

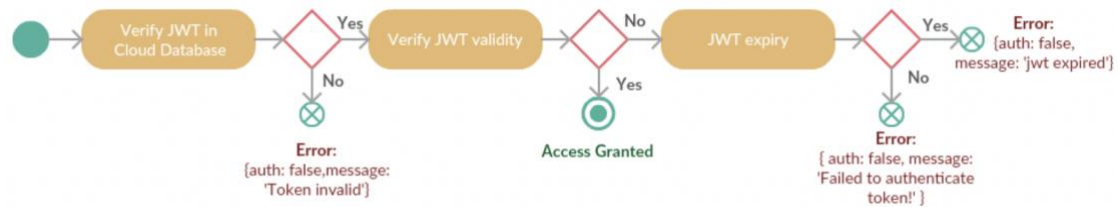


Figure 2 Authentication

When we attempt to send some data to the server over HTTPS request we have to first go through the authentication procedure that is already implemented. Doing that, the node trying to reach the server (TraCI in our case) can authenticate itself. As a reply of successful authentication, the client will receive a JWT using the “x-access-token” header of the response. The JWT must be stored in the client and used for further access to the Geo-server web services.

The subscription implementation of the data to the server had to be slightly modified because of the nature of the different attributes that we want to store. Of course new data base had to be added explicitly for our project.

2.5 Network map primer

The simulation and testing of the project is done in a random road in Sophia-Antipolis area. For our convenience we chose a small part of the map which contains two highway roads and we created two flows of five (5) cars in each direction. That way we could track the route of the cars more easily and understand how the extracted data are generated during the simulation. In *Figure 3* is presented the network on which we are working while in the *Figure 4* we can see the route-file that we used for this scenario.

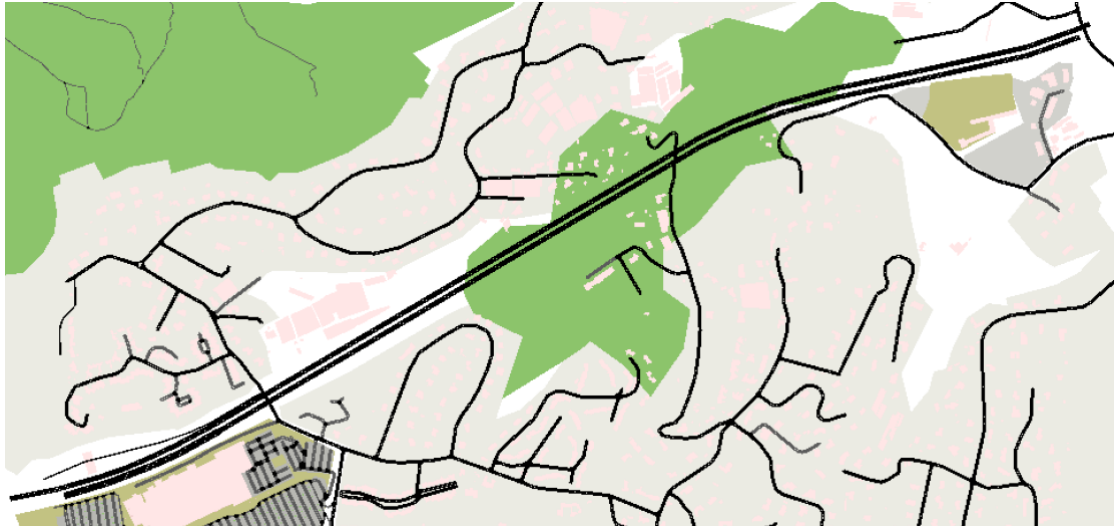


Figure 3 SUMO scenario network

```
<routes xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="http://
sumo.dlr.de/xsd/routes_file.xsd">

  <vType id="car" vClass="passenger" length="5" accel="3.5" decel="2.2" sigma="1.0" maxSpeed="60"/>

  <flow id="carflow" type="car" begin="0" end="0" number="5" from="197670283" to="8604402" />
  <flow id="carflow2" type="car" begin="0" end="0" number="5" from="45770917" to="521010125" />
</routes>
```

Figure 4 Flows generation

Although the scenario worked perfectly for this particular case it was desirable to be able to use and test the model at any other place we would like to study. So we created a bash file named “mapPrimer.sh” that generates the map network using the NETCONVERT SUMO’s library. All we need to do is provide the equivalent .osm file downloaded from openstreetmap.org and then modify the edges of the roads in the route-file to place a mobility flow according to the new network. The code can be reached at the map.rou.xml file inside the /scenario folder.

3. Implementation / Results

3.1 TraCI coding

The [TraCI](#) commands are split into the 13 domains gui, lane, poi, simulation, trafficlight, vehicletype, edge, inductionloop, junction, multientryexit, polygon, route, person and vehicle, which correspond to individual modules.

In our case we use the simulation domain to get all the IDs of the departed vehicles at every timestep. The TraCI loop runs for as long as there are vehicles running or expected to get imported in the simulation.

There are multiple ways to configure TraCI to extract information out of a simulation. But for the needs of our project we investigated two of them. The first one is to access data for every single vehicle of interest at each simulation looping through the whole simulation time, and the second is accessing data using subscriptions.

Subscriptions can be thought of as a batch mode for retrieving variables. Instead of asking for the same variables over and over again, you can retrieve the values of interest automatically after each time step. TraCI subscriptions are handled on a per module basis. That is you can ask the module for the result of all current subscriptions after each time step. In order to subscribe for variables you need to know their variable ids which can be looked up in the `traci/constants.py` file.

3.1.1 Subscription method

At every timestep we update the subscribed vehicles and we extract their current speed and their position as a combination of coordinates (latitude, longitude). It is critical to update the list as the cars are incoming into the simulation in different time steps so the list has to be updated dynamically.

When we have the list with the vehicles we add those values in a .txt file appending the new values in every timestep until the end of the simulation. That way we can have a detailed “history” of all the positions and speed of all the vehicles during the whole simulation. This file needs to be sent later to the geonet server in order to be manipulated in a meaningful way.

The block of code in *Figure 5* represents the above procedure.

```
103 while traci.simulation.getMinExpectedNumber() > 0:
104     for veh_id in traci.simulation.getDepartedIDList():
105         traci.vehicle.subscribe(veh_id, [traci.constants.VAR_POSITION]) # cant choose the second
            vehicle cause it hasnt departed yet. So the only solution is to subscribe for all the
            departed so far or just the first cause it instantly departs
106         traci.vehicle.subscribe(veh_id, [traci.constants.VAR_SPEED])
107
108     infos = traci.vehicle.getAllSubscriptionResults()
109     with open("mobility_data.txt", "a") as f:
110         json_infos = json.dumps(infos) # use `json.loads` to do the reverse
111         print(json_infos, file=f)
112     traci.simulationStep()
113     step += 1
114 traci.close()
115 sys.stdout.flush()
```

Figure 5 Traci script - Subscriptions

3.1.2 Simple iteration method

Following this method we keep the same logic as before (looping through all vehicles and timesteps) during the simulation but we store the data in a more custom way. This way we can create our own data structures that match our needs. On the negative side it is more computationally demanding and could slow down a lot the simulation in case of dense vehicle distribution. *Figure 6*

```

119 while traci.simulation.getMinExpectedNumber() > 0:
120     for veh_id in traci.simulation.getDepartedIDList():
121         vehicles_array.append(veh_id) # creating an array with the cars that have departed and so
122             are running in the simulation
123     for veh_id in traci.simulation.getArrivedIDList():
124         if veh_id in vehicles_array:
125             vehicles_array.remove(veh_id) # deleting the cars that have arrived to the
126                 destination. We need an updated array otherwise it will crash
127     for car in vehicles_array:
128         current_position = traci.vehicle.getPosition(car)
129         current_speed = traci.vehicle.getSpeed(car)
130         # print(current_position,current_speed)
131         with open("mobility_data.txt", "a") as output_file:
132             inputs = {
133                 "step": step,
134                 "ID": car,
135                 "position": current_position,
136                 "speed": current_speed
137             }
138             json_entry = json.dumps(inputs) # use `json.loads` to do the reverse
139             print(json_entry, file=output_file)
140     traci.simulationStep()
141     step += 1
142 traci.close()
143 sys.stdout.flush()

```

Figure 6 Traci script - Custom

3.2 JSON Output file

The collected data are saved in a .txt file named “mobility_infos.txt” in JSON format. The purpose of this file is to be sent to an external geonet server where after processing the containing data it would be possible to reconstruct the “running scene” and take decisions. For example if this file is sent to the geonet server every 30 seconds, the information that are enclosed will represent all the information (position and speed) of the running cars in a particular area for the measured time. Thus we will be able to compute any critical parameters such as traffic density and take any decisions accordingly (choose another route where there is less traffic for the incoming cars, regulate the adaptive speed limit, etc.)

In real world these kind of information would most probably be transmitted directly from the vehicles to an RSU (Road Side Unit) and then to the server every second. The actual procedure is still debatable but the notion remains the same.

It is critical for the sent data to be transmitted in a universal format for our ease and practicality. We chose the JSON format as we can handle the data easier. Of course in real world applications security encryption layers would be added on top of that.

JavaScript Object Notation (JSON) is an open-standard file format that uses human-readable text to transmit data objects consisting of attribute–value pairs and array data types (or any other serializable value). It is a very common data format used for asynchronous browser–server communication, including as a replacement for XML.

```
{
  'step': 1, 'ID': 'carflow.0', 'position': (183.96454899928068, 504.7425685135169), 'speed': 0.0
}, {
  'step': 1, 'ID': 'carflow2.0', 'position': (2052.276190359987, 1374.452221516854), 'speed': 0.0
}, {
  'step': 2, 'ID': 'carflow.0', 'position': (184.50514152645695, 504.9074195314196), 'speed': 1.1265683064993937
}, {
  'step': 2, 'ID': 'carflow2.0', 'position': (2051.829649209626, 1374.2621647462456), 'speed': 0.9703673738404177
}, {
  'step': 3, 'ID': 'carflow.0', 'position': (185.5463834929816, 505.22491672450815), 'speed': 2.16973046139943
}, {
  'step': 3, 'ID': 'carflow2.0', 'position': (2050.6062643255864, 1373.7414678741131), 'speed': 2.6585070069370165
}, {
  'step': 4, 'ID': 'carflow.0', 'position': (187.39059342950014, 505.78732382053437), 'speed': 3.843409751389535
}, {
  'step': 4, 'ID': 'carflow2.0', 'position': (2049.0526657425476, 1373.0802255019935), 'speed': 3.3760861139118976
}, {
  'step': 5, 'ID': 'carflow.0', 'position': (189.8667956022472, 506.5424294281578), 'speed': 5.160283851634139
}, {
  'step': 5, 'ID': 'carflow2.0', 'position': (2047.2923096376874, 1372.3309830276107), 'speed': 3.8253856987532533
}, {
  'step': 6, 'ID': 'carflow.0', 'position': (192.55703398121906, 507.3628043184487), 'speed': 5.6063248053182715
}, {
  'step': 6, 'ID': 'carflow2.0', 'position': (2045.1657589105941, 1371.4258808586414), 'speed': 4.621154047544424
}, {
  'step': 6, 'ID': 'carflow.1', 'position': (183.96454899928068, 504.7425685135169), 'speed': 0.0
}, {
  'step': 6, 'ID': 'carflow2.1', 'position': (2052.276190359987, 1374.452221516854), 'speed': 0.0
}, {
  'step': 7, 'ID': 'carflow.0', 'position': (195.69556095638615, 508.3198825993375), 'speed': 6.540536247856343
}, {
  'step': 7, 'ID': 'carflow2.0', 'position': (2042.2387578266614, 1370.1800911410996), 'speed': 6.3605926413394815
}, {
  'step': 7, 'ID': 'carflow.1', 'position': (184.61446945047115, 504.94075854121513), 'speed': 1.354402336786734
}, {
  'step': 7, 'ID': 'carflow2.1', 'position': (2051.7418684293416, 1374.2248035050504), 'speed': 1.1611215857847128
}, {
  'step': 8, 'ID': 'carflow.0', 'position': (199.62323793782957, 509.51760827225024), 'speed': 8.185086147183446
}, {
  'step': 8, 'ID': 'carflow2.0', 'position': (2038.962473960921, 1368.7856397436979), 'speed': 7.119610293880415
}, {
  'step': 8, 'ID': 'carflow.1', 'position': (185.45197480204135, 505.1961516566665), 'speed': 1.7453200667255664
}, {
  'step': 8, 'ID': 'carflow2.1', 'position': (2050.5794127914264, 1373.7300393285823), 'speed': 2.526103190393872
}, {
  'step': 9, 'ID': 'carflow.0', 'position': (204.3518716998065, 510.9595817546113), 'speed': 9.854240784851262
}, {
  'step': 9, 'ID': 'carflow2.0', 'position': (2035.1401606271986, 1367.1587873743863), 'speed': 8.306173235406478
}, {
  'step': 9, 'ID': 'carflow.1', 'position': (186.69904314779026, 505.5764389818811), 'speed': 2.5988292544437286
}, {
  'step': 9, 'ID': 'carflow2.1', 'position': (2049.162162320057, 1373.1268294172144), 'speed': 3.079791452284656
}, {
  'step': 10, 'ID': 'carflow.0', 'position': (209.13169742931797, 512.4171659743249), 'speed': 9.960922333841571
}, {
  'step': 10, 'ID': 'carflow2.0', 'position': (2030.710702103961, 1365.4205294290712), 'speed': 9.518811663142715
}, {
  'step': 10, 'ID': 'carflow.1', 'position': (188.0936493107464, 506.00171723487045), 'speed': 2.9062908276621897
}, {
  'step': 10, 'ID': 'carflow2.1', 'position': (2047.4497467606027, 1372.3979913898434), 'speed': 3.721207301961943
}, {
  'step': 11, 'ID': 'carflow.0', 'position': (214.29418649105605, 513.99144148491), 'speed': 10.758373945682669
}, {
  'step': 11, 'ID': 'carflow2.0', 'position': (2026.1306237253661, 1363.7981160831637), 'speed': 9.71546766260437
}, {
  'step': 11, 'ID': 'carflow.1', 'position': (190.12922272807393, 506.6224552824621), 'speed': 4.242035141499684
}, {
  'step': 11, 'ID': 'carflow2.1', 'position': (2045.2513890247883, 1371.4623267336908), 'speed': 4.777196057155482
}, {
  'step': 11, 'ID': 'carflow2.2', 'position': (2052.276190359987, 1374.452221516854), 'speed': 0.0
}
```

Figure 7 JSON file

In [Figure 7](#) we can see what the actual JSON file looks like. This particular part corresponds to the begin of the simulation until some seconds later. Every timestep where we have our measurements is represented with a pair of curly braces. As we can see the timestamp=0 - which is the very first moment of the simulation - we have no data inside curly braces as there are still no vehicles running in the roads. if we read the file more carefully we can observe the evolution of the traffic.

We can have a closer look at the JSON string by investigating the below example.

```
{ "step": 1, "ID": "carflow.0", "position": [183.96454899928068, 504.7425685135169], "speed": 0.0 }
```

Timestamp = 1. This is the first measurement of the simulation and as we can see the two vehicles have been inserted in the simulation, one at the east edge and one at the west edge. They have position attributes but their speed attribute is still zero.

The following JSON script represents the above state while in [Figure 8](#) we can see a more human-friendly representation of the same code.

```
{
  "step": 1,
  "ID": "carflow.0",
  "position": [
    183.96454899928068,
    504.7425685135169
  ],
  "speed": 0.0
}
```

Figure 8 JSON entry

The screenshot from the SUMO simulation in [Figure 9](#) shows how the flow is generated while cars are being inserted into the simulation so we can have a better understanding reading the JSON file. It should be noted that the screenshot does not correspond to any of the examples given.

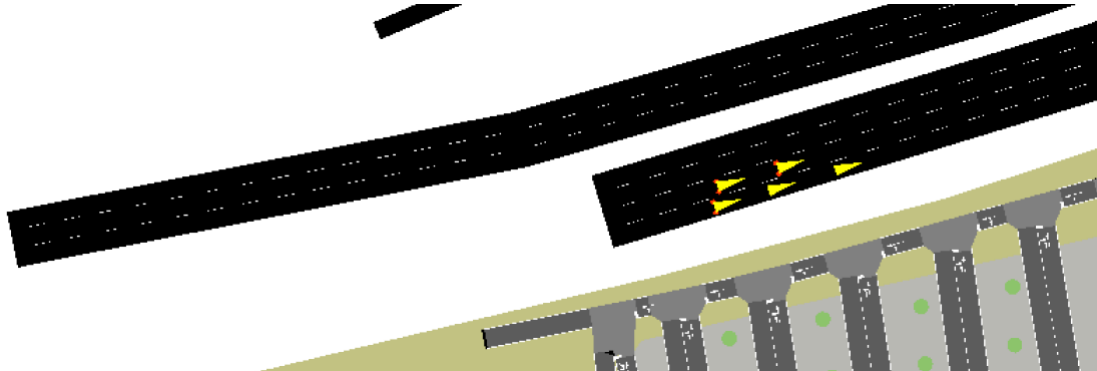


Figure 9 Vehicle flow west-east

In case that we generate the file using subscriptions the data structure is a bit different as the information are automatically transferred and stored to the file.

Each vehicle has its own ID so that we can keep track of the individual routes and behaviors. The position of the vehicle is given after the code "66" while the speed is given with the code "64". The overall structure of the file is pretty robust as every information is given followed by codes and so the manipulation of the data could easily be done with a decomposition script as part of the Geo-server server.

We can have a closer look at the JSON string by investigating the below example.

`{"carflow.0": {"66": [2052.276190359987, 1374.452221516854], "64": 0.0}, "carflow2.0": {"66": [183.96454899928068, 504.7425685135169], "64": 0.0}}`. [Figure 10](#)

```
{
  "carflow.0": {
    "66": [
      2052.276190359987,
      1374.452221516854
    ],
    "64": 0.0
  },
  "carflow2.0": {
    "66": [
      183.96454899928068,
      504.7425685135169
    ],
    "64": 0.0
  }
}
```

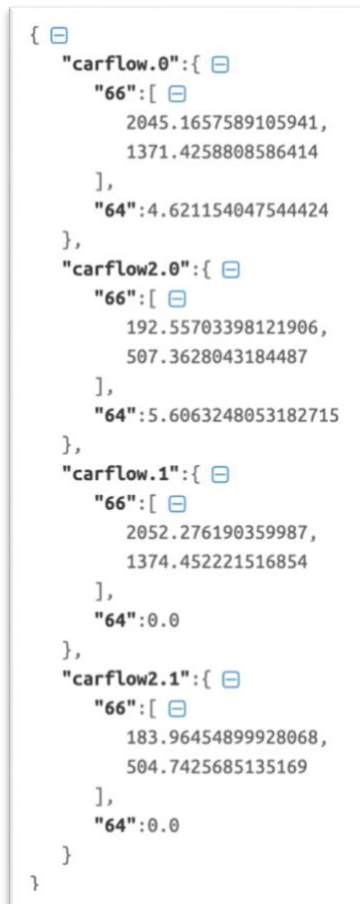
Figure 10 Timestamp = 1

Timestamp = 7. This is the seventh measurement of the simulation and as we can deduce four vehicles have been inserted in the simulation, two at the east edge and two at the west edge. Two of them had already been inserted in the simulation in a previous timestep so

they have some value greater than zero as speed and the other two just got in the simulation so speed is zero.

The following JSON script represents the above state while in *Figure 11* we can see a more human-friendly representation of the same code.

```
{ "carflow.0": { "66": [2045.1657589105941, 1371.4258808586414], "64": 4.621154047544424}, "carflow2.0":
{"66": [192.55703398121906, 507.3628043184487], "64": 5.6063248053182715}, "carflow.1": {"66":
[2052.276190359987, 1374.452221516854], "64": 0.0}, "carflow2.1": {"66": [183.96454899928068,
504.7425685135169], "64": 0.0}}
```



```
{
  "carflow.0": {
    "66": [
      2045.1657589105941,
      1371.4258808586414
    ],
    "64": 4.621154047544424
  },
  "carflow2.0": {
    "66": [
      192.55703398121906,
      507.3628043184487
    ],
    "64": 5.6063248053182715
  },
  "carflow.1": {
    "66": [
      2052.276190359987,
      1374.452221516854
    ],
    "64": 0.0
  },
  "carflow2.1": {
    "66": [
      183.96454899928068,
      504.7425685135169
    ],
    "64": 0.0
  }
}
```

Figure 11 Timestamp = 7

3.3 Connection to the Geo-server and data storage

As mentioned before, the purpose of the whole simulation is the generation of the vehicle mobility data is to use them in subsequent time in order to get informed for the surrounding mobility situation of a particular area. The most efficient way to achieve that is to gather all the data in a remote server (Geo-server), where we can apply especially for this case developed techniques, with the ultimate goal the reconstitution of the traffic network. Consequently, we will be able to create dedicated snapshots of the map that correspond to any time stamp of the simulation we want.

The implementation of the postage and storage of the generated data could vary as there are many ways to proceed. The initial thought was to create a JSON file that would contain all the information and to send this file over a single HTTPS connection to the Geo-server.

There a dedicated script would be responsible for the decomposition of the file to actual usable data. Picturing this in a real world case scenario, we could say that TraCI has the role of side road infrastructure which collects all the vehicle information, stores them and forwards them to the remote server. Because of the limited time and the complexity of the backend part of the server in a case like that, we decided to turn to a more feasible solution for the time being in order to have some solid results.

In first phase, we decided to send the data over an HTTPS request during the actual simulation time. This means that TraCI triggers an HTTPS request to the server in every iteration of the simulation and the data are being stored in a server data base in real time. In the real world that would correspond to the actual vehicles sending their measured information to the road infrastructure every a specific amount of time. This method gives us access to the actual data that are instantly stored in a database much faster than the first method. As expected, the huge load of HTTPS connections has negative impact on the simulation time and further modifications should be made to the code, in order to be less computationally demanding. For example the frequency of data sampling could be lower, but more tests should be done to proceed in such decisions.

In any case both methods have their advantages and disadvantages but the best way of implementation is still debatable and under continuous development.

In the Figure 12 is represented the block of communication of TraCI with the server while in *Figure 13* we can see the actual data of the database after a GET request.

```

46     for car in vehicles_array:
47         current_position = traci.vehicle.getPosition(car)
48         current_speed = traci.vehicle.getSpeed(car)
49         # print(current_position, current_speed)
50         mydata = {
51             "step": step,
52             "ID": car,
53             "position": current_position,
54             "speed": current_speed
55         }
56
57         headers = {'Content-Type': 'application/json', 'Accept': 'text/plain'}
58         data_json = json.dumps({"mydata":mydata})
59         resp = requests.post("https://geoserver.eurecom.fr/sumo", data = data_json, headers = headers)
60
61         data_json_txt = json.dumps(mydata)
62         with open("mobility_data.txt", "a") as output_file:
63             # print("data_json_txt", file=output_file) # print needs to be changed to pyhon2.7
64             infos = traci.vehicle.getAllSubscriptionResults()
65             with open("mobility_infos.txt", "a") as f:
66                 json_infos = json.dumps(infos) # use 'json.loads' to do the reverse
67                 # print(json_infos, file=f) # print needs to be changed to pyhon2.7
68             traci.simulationStep()
69             step += 1
70         traci.close()
71         sys.stdout.flush()

```

Figure 12 Communication with Geoserver

```
{
  "sumo_data": [
    {
      "id": "5d122d487f95be5b9dab20f8",
      "step": "1",
      "ID": "2",
      "position": "3,4",
      "speed": "5km/hr",
      "id": "5d122dbe7f95be5b9dab20f9",
      "step": "1",
      "ID": "2",
      "position": "3,4",
      "speed": "5km/hr",
      "id": "5d122de67f95be5b9dab20fa",
      "step": "1",
      "ID": "2",
      "position": "3,4",
      "speed": "5km/hr",
      "id": "5d122eba1b937a787351f882",
      "step": "1",
      "ID": "2",
      "position": "3,4",
      "speed": "5km/hr",
      "id": "5d122ef1b937a787351f883",
      "step": "1",
      "ID": "2",
      "position": "3,4",
      "speed": "5km/hr",
      "id": "5d12307c4e3f541c61531b2e",
      "step": "dummy_step",
      "ID": "dummy_car",
      "position": [1335.140160627, 1986.1367.1587873743863],
      "speed": "dummy_speed",
      "id": "5d1232404e3f541c61531b2f",
      "step": "1",
      "ID": "carflow.0",
      "position": [183.96454899928068, 504.7425685135169],
      "speed": "0",
      "id": "5d1232404e3f541c61531b30",
      "step": "1",
      "ID": "carflow2.0",
      "position": [2052.276190359987, 1374.452221516854],
      "speed": "0",
      "id": "5d1232404e3f541c61531b31",
      "step": "2",
      "ID": "carflow.0",
      "position": [184.50514152645695, 504.9074195314196],
      "speed": "1.1265683064993937",
      "id": "5d1232414e3f541c61531b32",
      "step": "2",
      "ID": "carflow2.0",
      "position": [2051.829649209626, 1374.2621647462456],
      "speed": "0.9703673738404177",
      "id": "5d1232414e3f541c61531b33",
      "step": "3",
      "ID": "carflow.0",
      "position": [185.5463034929816, 505.22491672450815],
      "speed": "2.16973046139943",
      "id": "5d1232424e3f541c61531b34",
      "step": "3",
      "ID": "carflow2.0",
      "position": [2050.6062643255864, 1373.7414678741131],
      "speed": "2.6585070069370165",
      "id": "5d1232424e3f541c61531b35",
      "step": "4",
      "ID": "carflow.0",
      "position": [187.39059342950014, 505.78732382053437],
      "speed": "3.843409751389535",
      "id": "5d1232434e3f541c61531b36",
      "step": "4",
      "ID": "carflow2.0",
      "position": [2049.0526657425476, 1373.0802255019935],
      "speed": "3.3760861139118976",
      "id": "5d1232434e3f541c61531b37",
      "step": "5",
      "ID": "carflow.0",
      "position": [189.8667956022472, 506.5424294281578],
      "speed": "5.160283851634139",
      "id": "5d1232444e3f541c61531b38",
      "step": "5",
      "ID": "carflow2.0",
      "position": [2047.2923096376874, 1372.3309830276107],
      "speed": "3.8253856987532533",
      "id": "5d1232444e3f541c61531b39",
      "step": "6",
      "ID": "carflow.0",
      "position": [192.55703398121906, 507.3628043184487],
      "speed": "5.6063248053182715",
      "id": "5d1232454e3f541c61531b3a",
      "step": "6",
      "ID": "carflow2.0",
      "position": [2045.1657589105941, 1371.4258808586414],
      "speed": "4.621154047544424",
      "id": "5d1232454e3f541c61531b3b",
      "step": "6",
      "ID": "carflow.1",
      "position": [183.96454899928068, 504.7425685135169],
      "speed": "0",
      "id": "5d1232454e3f541c61531b3c",
      "step": "6",
      "ID": "carflow2.1",
      "position": [2052.276190359987, 1374.452221516854],
      "speed": "0",
      "id": "5d1232464e3f541c61531b3d",
      "step": "7",
      "ID": "carflow.0",
      "position": [195.69556095638615, 508.3198825993375],
      "speed": "6.540536247856343",
      "id": "5d1232474e3f541c61531b3e",
      "step": "7",
      "ID": "carflow2.0",
      "position": [2042.2387578266614, 1370.1800911410996],
      "speed": "6.3605926413394815",
      "id": "5d1232474e3f541c61531b3f",
      "step": "7",
      "ID": "carflow.1",
      "position": [184.61446945047115, 504.94075854121513],
      "speed": "1.354402336786734",
      "id": "5d1232474e3f541c61531b40",
      "step": "7",
      "ID": "carflow2.1",
      "position": [2051.7418684293416, 1374.2248035050504],
      "speed": "1.1611215857847128",
      "id": "5d1232484e3f541c61531b41"
    ]
  ]
}
```

Figure 13 Database data

4. Conclusion

The ability to monitor vehicle behaviors and traffic flows can lead to novel solutions for a smarter, more efficient and ecological friendly mobility in modern cities and highways. Vehicular data management and data exploitation are key factor to transfer all the on-road information to a remote server and acknowledge the actual reality conditions.

This aim of this project was to create a simple scenario which represents a vehicular traffic and extract the some information like position and speed sending them to a remote server playing the role of a road infrastructure node. Although at a very early stage the goal was achieved and we managed to end up with a database that consists of the accurate positions and speeds. The project has to be continued and more work has to be done both on the data management and the structural data interpretation in order to develop a solid framework that handles the vehicular data metrics efficiently and solidly.

The chosen network, although it is very simple serves perfectly our demands, but in reality more complex network could bring unexpended problems as multiple routes directions and physical parameters (traffic lights, traffic signage, etc.) would bring the need of adding new logics to the traffic detection algorithms.

On the Geo-server part we managed to have a working database where we can store the collected information an use them according to our needs. The database management should be defined in a later phase as well as possible changed in the structure of database.

5. Future Development

As this project constitutes the very first phase of continuous and larger project for traffic jam detection more work is expected to be done in the near future. Below are pointed some recommendations for future development, based on the current problems or future malfunctions that could be faced later on.

- Define the most optimal way to send data from TraCI client to the Geo-server. Study on the impact of multiple requests and explore possible solutions to create file or object using a single request.
- Define the procedure of deleting the deprecated database entries. There is no need to keep any vehicular information for long time as all the information concern real time conditions
- Develop the algorithm of traffic jam detection using the data of the database and the traffic jam detecting techniques that Romain developed in his part.
- Integrate Vanetza stack to handle the postage of the vehicular data as CAM messages.

References

- [1] https://sumo.dlr.de/wiki/Simulation_of_Urban_MObility_-_Wiki
- [2] <https://sumo.dlr.de/wiki/TraCI>
- [3] http://sumo.sourceforge.net/userdoc/TraCI/Interfacing_TraCI_from_Python.html
- [4] <https://github.com/eclipse/sumo/tree/master/tools/traci>
- [5] <https://github.com/riebl/vanetza>
- [6] <https://en.wikipedia.org/wiki/JSON>
- [7] <https://jsonformatter.curiousconcept.com>

Appendices

```
1  #!/bin/bash
2
3  # exit on error
4  set -e
5
6  export SUMO_HOME="$home/Users/alex/Desktop/sumo"
7  export SUMO_BINS="$home/Users/alex/Desktop/sumo/bin"
8  export SUMO_TOOLS="$home/Users/alex/Desktop/sumo/tools"
9
10 ROOT="."
11 DATA="/Users/alex/Desktop/proj/data"
12 SCRIPTS="/Users/alex/Desktop/proj/scripts"
13 SCENARIO="/Users/alex/Desktop/proj/scenario"
14 TAZ="$SCENARIO/taz"
15
16
17
18 echo "Creating the network..."
19 $SUMO_BINS/netconvert -c $SCENARIO/map.netcfg
20 netconvert --osm-files $DATA/map.osm --lefthand --output.street-names -o map.net.xml
21
22 echo "Extracting the polygons..."
23 $SUMO_BINS/polyconvert -c $SCENARIO/map.polycfg
24
25 echo "Running SUMO..."
26 $SUMO_BINS/sumo-gui -c $SCENARIO/map.sumocfg
```

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <configuration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4    xsi:noNamespaceSchemaLocation="http://sumo.dlr.de/xsd/sumoConfiguration.xsd">
5
6    <input>
7      <net-file value="map.net.xml"/>
8      <route-files value="map.rou.xml"/>
9      <additional-files value="map.poly.xml"/>
10    </input>
11
12    <output>
13      <summary-output value="summary.xml"/>
14      <tripinfo-output value="tripinfo.xml"/>
15      <vehroute-output value="vehroute.xml"/>
16      <stop-output value="ptoutput.xml"/>
17    </output>
18
19    <time>
20      <begin value="0"/>
21      <step-length value="0.5"/>
22      <end value="10000"/>
23    </time>
24
25    <report>
26      <verbose value="true"/>
27      <no-step-log value="true"/>
28    </report>
29
30    <traci_server>
31      <remote-port value="8888"/>
32      <num-clients value="1" type="INT" help="Expected number of connecting clients"/>
33    </traci_server>
34  </configuration>
```

```

1 #!/usr/bin/env python
2 import os
3 import sys
4 import optparse
5 import subprocess
6 import json
7 import requests
8
9 # we need to import some python modules from the $SUMO_HOME/tools directory
10 if 'SUMO_HOME' in os.environ:
11     tools = os.path.join(os.environ['SUMO_HOME'], 'tools')
12     sys.path.append(tools)
13     # tools = os.path.join(os.environ['HOME'], 'Desktop', 'sumo', 'tools')
14     # sys.path.append(tools)
15     # print(tools)
16
17 else:
18     sys.exit("please declare environment variable 'SUMO_HOME'")
19
20 from sumolib import checkBinary # Checks for the binary in environ vars
21 import traci
22
23 PORT = 8888
24
25 # contains TraCI control loop
26 def run():
27     """execute the TraCI control loop"""
28     traci.init(PORT)
29     step = 0
30     vehicles_array = []
31     # CLEAR THE FILE FROM PREVIOUS INPUTS
32     open("mobility_infos.txt", "w").close()
33     open("mobility_data.txt", "w").close()
34
35     while traci.simulation.getMinExpectedNumber() > 0:
36         for veh_id in traci.simulation.getDepartedIDList():
37             # vehicles_array = []
38             vehicles_array.append(veh_id) # creating an array with the cars that have
39             # departed and so are running in the simulation
40             traci.vehicle.subscribe(veh_id, [traci.constants.VAR_POSITION]) # cant choose
41             # the second vehicle cause it hasnt departed yet. So the only solution is to
42             # subscribe for all the departed so far or just the first cause it instantly
43             # departs
44             traci.vehicle.subscribe(veh_id, [traci.constants.VAR_SPEED])
45         for veh_id in traci.simulation.getArrivedIDList():
46             if veh_id in vehicles_array:
47                 vehicles_array.remove(veh_id) # deleting the cars that have arrived to the
48                 # destination. We need an apdated array otherwise it will crash
49         for car in vehicles_array:
50             current_position = traci.vehicle.getPosition(car)
51             current_speed = traci.vehicle.getSpeed(car)
52             # print(current_position,current_speed)

```

```

47 #         print(current_position,current_speed)
48         mydata = {
49             "step": step,
50             "ID": car,
51             "position": current_position,
52             "speed": current_speed
53         }
54
55         headers = {'Content-Type': 'application/json', 'Accept': 'text/plain'}
56         data_json = json.dumps({"mydata":mydata})
57         resp = requests.post("https://geoserver.eurecom.fr/sumo", data = data_json,
58                               headers = headers)
59
60         traci.simulationStep()
61         step += 1
62         traci.close()
63         sys.stdout.flush()
64
65     def get_options():
66         opt_parser = optparse.OptionParser()
67         opt_parser.add_option("--nogui", action="store_true", default=False, help="run the
68             cmdline version of sumo")
69         options, args = opt_parser.parse_args()
70         return options
71
72     # main entry point
73     if __name__ == "__main__":
74         options = get_options()
75
76         # check binary
77         if options.nogui:
78             sumoBinary = checkBinary('sumo')
79         else:
80             sumoBinary = checkBinary('sumo-gui')
81
82         # this is the normal way of using traci. sumo is started as a
83         # subprocess and then the python script connects and runs
84         sumoProcess = subprocess.Popen([sumoBinary, "-c", "scenario/map.sumocfg", "--tripinfo-
85             output", "tripinfo.xml"], stdout=sys.stdout, stderr=sys.stderr)
86         run()
87         sumoProcess.wait()

```

```

1  #!/usr/bin/env python
2
3  import os
4  import sys
5  import optparse
6  import subprocess
7  import json
8  import requests
9
10
11  mydata = {
12      "step": "dummy_step",
13      "ID": "dummy_car",
14      "position": [2035.1401606271986, 1367.1587873743863],
15      "speed": "dummy_speed"
16  }
17
18  headers = {'Content-Type': 'application/json', 'Accept': 'text/plain'}
19
20  data_json = json.dumps({"mydata":mydata})
21  print(data_json)
22  resp = requests.post("https://geoserver.eurecom.fr/sumo", data = data_json, headers =
    headers)
23
24
25
26  if resp:
27      print('Success!')
28  else:
29      print('An error has occurred.')

```

```

1  #!/usr/bin/env python
2  #!/usr/bin/python3
3
4
5  import os
6  import sys
7  import optparse
8  import subprocess
9  import json
10 import requests
11
12 resp = requests.get("https://geoserver.eurecom.fr/sumodatabasedump")
13 print(resp.text)
14 #print(resp.json)
15 #print(resp.content)
16 #print(resp.status_code)
17 #print(resp.history)
18 #print(resp.url)
19
20 if resp:
21     print('Success!')
22 else:
23     print('An error has occurred.')

```