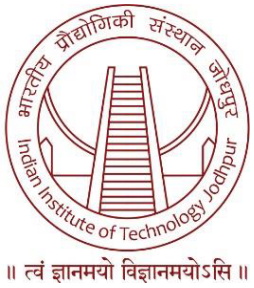
Edge-based knowledge inference for intelligent vehicular networks

Mentor: Dr. Suchetna Chakraborty

Department of Computer Science and Engineering
Indian Institute of Technology, Jodhpur

Bachelor of Technology, Final Year Project



Submitted by -
Rituraj Kulshresth (B18CSE046)
Saurabh Burewar (B18CSE050)

1. Introduction

With advancement in technology in the field of vehicular networks and autonomous vehicles, we have reached a point where we have many technologies catering to the specific needs of such networks like the WAVE architecture. But, there are many other problems which make it difficult to implement such networks and make our vehicles connected and able to communicate with each other.

One problem that comes with vehicular networks is the large amount of data. Handling large amounts of data has become a very important aspect of any large project in the modern world and the same is the case with vehicular networks. With millions of vehicles on the roads, the amount of data that these vehicles can generate is huge which makes handling of this data crucial. There are some ways in which we can tackle this problem. One is to improve the handling of this data at RSUs and at stations to make sure we use the data efficiently. Another solution is to reduce the amount of data itself without losing information, so that we have less data to deal with at RSUs and stations.

In this project, we have tried to experiment and create a filtering policy that can help reduce the amount of data sent by vehicles based on different conditions. This helps us in not only reducing the amount of data we have to deal with but also has other benefits. We save the bandwidth it is transferred on and also reduce the processing power required to process the data at RSUs and stations. A more detailed approach to this can be found in the Experiments section of this report.

2. Objective

The objective of this project is to reduce the amount of data transferred between vehicles and Road-Side Units (RSUs), so as to reduce the processing power required at the RSUs and make it possible for the architecture to perform the required calculations irrespective of the network strength.

3. Challenges/issues

Ns-3 is a network simulator built to experiment with various Internet Systems. The most challenging part of the project was to get proper guiding reference for the various parts of NS-3 code. The NS-3 documentation does not give examples for any of its code which makes the interpretation of the results very difficult. Moreover some of the latest versions of NS-3 had broken code which made working with the updated features impossible.

4. Experiments

To experiment with the filtering techniques to reduce data transfer, we created a simulation on NS-3 (Network Simulator 3) using OpenStreetMap and SUMO to get a realistic vehicular network. We implemented the filtering policy on the simulation to test its performance. A detailed explanation of our approach and experiment is given below. Here node refers to the vehicles that are generated in the scenario.

1. Creating a simulation

We created a custom simulation using OpenStreetMap and SUMO. The scenario was generated keeping in mind the capabilities of our system and the time taken by the application to run over the scenario. We created the simulation using the map of Berlin covering an area of approximately 2.5km X 2.0km. In order to create the NS-3 readable mobility file we follow these steps:

- First the trace file is extracted from the sumo config file which contains all the details of the map and the necessary items to recreate the simulation using the below command:

```
sumo -c osm.sumocfg --fcd-output trace.xml
```

- Next we use the trace exporter file of SUMO to generate the mobility.tcl file which contains the mobility of each node/vehicle at each second of the simulation:

```
python traceExporter.py -i 2022-01-06-20-47-08/trace.xml
--ns2mobility-output=/home/.../mobility.tcl
```

Now we can use this trace file in NS-3 in our custom application.

Here is the map of Berlin that we are using for the simulation. 52°31'07.3"N 13°23'07.8"E.

2. Nodes

The nodes in the mobility.tcl file are present in the following form:

```
$node_(0) set X_ 1441.62
$node_(0) set Y_ 1055.42
$node_(0) set Z_ 0
$ns_ at 0.0 "$node_(0) setdest 1441.62 1055.42 0.00"
```

Each node can be created at any place in the file before their use, but they must be initialized with a *setdest* statement at the time that the node has to start working in the simulation.

It is to be noted that the nodes have all been created as soon as the .tcl file is loaded however they will only start working or sending/receiving once their duration of existence starts and will end the sending/receiving once their duration ends. Both the start and end of the duration of existence of the node must be marked by time, i.e. there must be a *setdest* line from the time when the nodes start working to the time when the nodes must end working. However, the nodes are not removed from the simulation when their lifespan ends. They remain there as stationary nodes.

- The 0.0 marks the start time of the node

  ```
  $ns_ at 0.0 "$node_(0) setdest 1441.62 1055.42 0.00"
  ```

- The two floating point numbers mark the destination of the node (Here it's a stationary node/RSU hence the initialization position and destination are the same).

  ```
  $ns_ at 0.0 "$node_(0) setdest 1441.62 1055.42 0.00"
  ```

- The 0.00 marks the speed of movement of the node on the map towards the destination.

  ```
  $ns_ at 0.0 "$node_(0) setdest 1441.62 1055.42 0.00"
  ```

- This statement marks the end of lifespan as at 24 sec there is no *setdest* definition for the node (in the chosen example).

  ```
  $ns_ at 24.0 "$node_(0) setdest 1661.04 1032.11 13.46"
  ```

The RSUs are added to the .tcl file manually by copying the above snippet from other nodes. The RSUs are placed at a distance of 180 units from each other since the Wave devices can send the data up to 180 units. The starting time of the RSUs is set up to 0 and the end time of the RSUs is set to the simulation time in seconds. The number of RSUs depends on the distribution of the nodes/vehicles across the map. Here we created 36 RSUS in a 4x9 grid over the map to facilitate the communication of the nodes with the RSUs.

3. Communication between nodes

   The communication between the nodes is set up using WAVE Short Message Protocol (WSMP). Each node is first added to a node container. Then a NetDevice container is created and linked to this node container, such that each node gets a WaveNet device of its own. This device is the identity that lets the nodes communicate with each other as and when required.

   Then a Custom Application is created in which based on the conditions of the use case the nodes communicate with each other.

4. Creating packets and tags

   Nodes communicate with each other through WAVE SM protocol by sending packets. A packet consists of a byte buffer, byte tags, packet tags and metadata. The metadata describes the headers and trailers used to serialize data in the buffer.

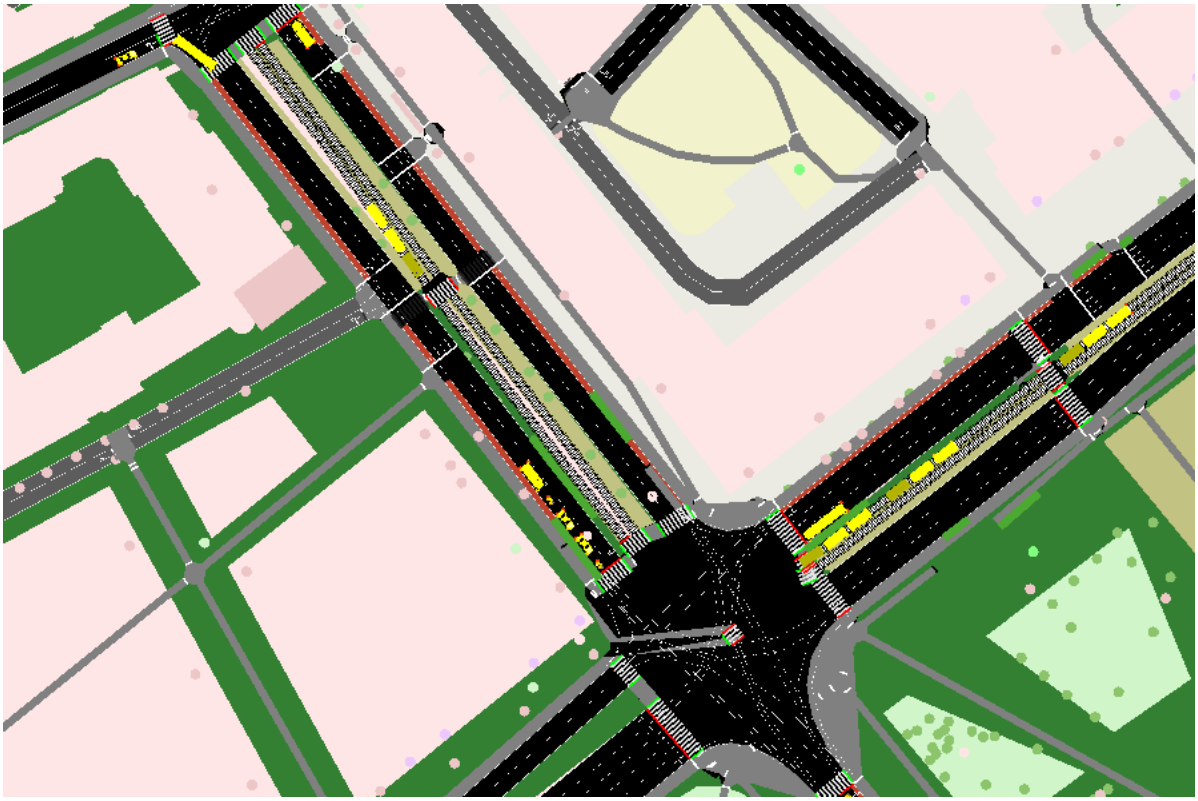
   There are two methods in which data can be sent using packets -

   - We can store the data in the byte buffer of the packet. Doing this, however, requires headers to be created so that the data in the buffer can be serialized and also requires the writing of our own print function to print the data in the buffer, since there is no method in NS-3 that enables printing the buffer contents.
   - The second method is the use of packet tags. Packet tags are attached to and travel with the packets. We can write custom serialization and printing functions for our tags to access the contents of the packet tags easily.

   In this experiment, we have used packet tags to send data in packets.

   The packets and tags are created every 0.1 seconds for every node. First, we create a packet with a 1000 bytes size byte buffer (we are not using the buffer here though). Then, we create the tag and get the node id, node position and node velocity and add it to the tag. All the information is stored in serialized form in the buffer inside the tag. Once the packet is ready, we send it or drop it based on our filtering policy discussed below.

5. Filtering



Above is a zoomed in map of the simulation showing all the vehicles to measure and judge the scale of the map and vehicles in order to decide the filtering conditions. We are filtering the packets by change in position and velocity and also make sure that packets are not dropped consecutively for a long time. Below are the conditions used to decide whether the packet should be sent or dropped.

- First, the amount of change that occurred in the velocity of the nodes, as compared to the previous packet is checked.

  We want to look for a considerable change in velocity. An instant stop of a vehicle occurs when the change velocity is equal to the initial velocity (since final velocity is 0). Since we want to look for considerable changes and not just drastic changes, we can say the change in velocity to be around half of the initial velocity. To generalize this metric, we can use the average velocity by which a vehicle travels. From running the simulation, we calculated that the average velocity with which the vehicles travel is around 0.53 meters per second. Therefore, a considerable change in velocity is when there is a change of 0.26 meters per second and then the packet is sent.

- If the above condition is not satisfied, the amount of change that occurred in the position (coordinates) of the nodes, as compared to the previous packet is checked.

  The simulation created by SUMO scales such that 1 unit on the map is equal to 1 meter in the real world. Furthermore, the average car length (length of a vehicle - car, bus, etc.) in the simulation is equal to 4.83 units. Since a major change in position of a vehicle is a change equal to the length of the vehicle, we decided to go for a threshold equal to one car length. Therefore, if the change in position is greater than 4.83 meters, then the packet is sent.

- If the figures don't satisfy the above conditions, we check when the last packet was sent.

  Continuously, not sending packets for a long time will cause loss of data and to prevent that we check for the last sent packet. If a node/vehicle drops 5 packets consecutively, we send the next packet irrespective of the change in its position or velocity, to prevent loss of data for that node. Here 5 is selected as a random number for the experiment and can be changed based on the map size and simulation details.

- If none of the above conditions are satisfied, then the packet is dropped.

It is to be noted that these conditions are set with the current map in mind and there may be a need to change the thresholds if the map size increases or the density of vehicles changes or if some other variable is introduced in the simulation.

```
Payload (size=1000)
0x555562b1e2d0
    Previous pos: 1368.52:670.781:0 Current pos: 1368.51:670.779:0
    DIfference: 0.00263969

Sent: 303696 out of 1127664
```

Here as we can see the application sends out 303696 packets out of the total 1127664 packets, which is roughly around 1/4th. The rest of the packets are dropped due to the above mentioned filtration process.

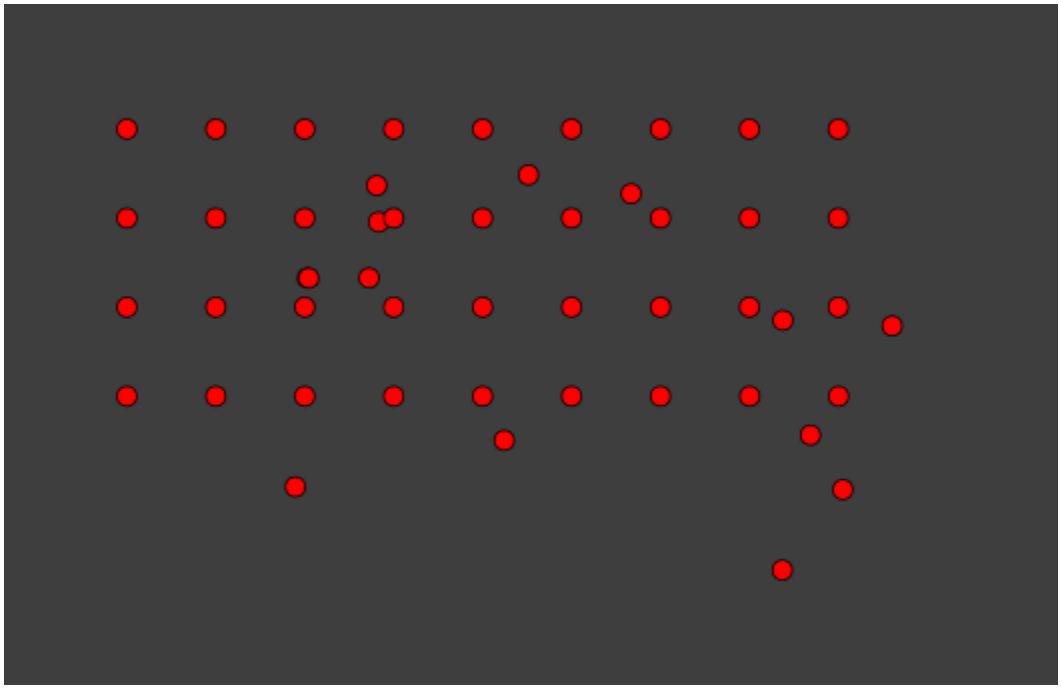
6. Sending and receiving packets

Based on the above filtering conditions, we decide whether to send our packets or drop them.

To send the packets, we use the SendX function which lets us send packets to a specific MAC address. In this simulation, we already know the addresses of all the wavenet devices of the RSUs and so we can set them beforehand and send the packets to those addresses only. But this causes a problem in NS-3. When sending the packet to multiple nodes, NS-3 tries to duplicate the tag attached to the packet which is not possible in NS-3. To solve this, we modified the wavenet device class of NS-3 and created another function called *SendXe* which sends the packet without adding the tag every time.

So we first use the SendX function for the first RSU and now the tag has been attached to the packet, hence for the next addresses of the RSUs we use the SendXe function. We send packets to all the RSUs using the *SendX* and *SendXe* functions in the above order.

At the receiving end, the packets are received with the application data and no WifiMacHeader. The contents of the packet are checked along with whether there is a tag attached. The tag contents are acquired and printed on the terminal.



The above image shows a visualization of the network created by NS-3. Here, we can see all the nodes being depicted in red and as we run the simulation, these nodes move in the same way as in the SUMO simulation.

5. Application

The application built for this project starts from the main function in the wave-project.cc file which loads the nodes using the *NS2NodeUtility* class.

The *NS2NodeUtilty* class is built to facilitate the reading of node movements and positions throughout the simulation and based upon this data other parts of the applications start their work. This class reads the data and counts the number of nodes and the simulation time for the nodes. It also collects all the details regarding the time of entry and exit of a node in the simulation.

Next the *NodeContainer* creates the nodes and stores their details. In order to give mobility to these nodes according to the data present in the tcl file, the *Ns2MobilityHelper* reads the sumo trace and installs it on the nodes.

Next we create the WaveNet devices. This is done using the *WaveSetup* class. This class takes a node container as input. It sets the WAVE device with the default MAC and Physical layer settings. The datalink type is IEEE 802.11p. We set the transmitted power to 33. The rate of transmission is set to constant to 6Mbps, the datamode is set to OFDM and the bandwidth is set to 10MHZ. With all these settings the devices are created and are installed on the nodes which are then returned.

Now the application part is started which handles most of the logic. In the application part, we define the time duration between each broadcast and the packet size in which the custom data of the node is stored. Here this custom data is a random value. It has been left as it is with a size of 1000 bytes so that any details regarding the vehicle's make, model, driver details, passenger details can be put into the packet and sent to the RSU. Once the application is started it iterates over each node present in the node container and sets up callbacks for the time whenever a packet is sent or received by a node.

Whenever a node receives a packet it first calls the promiscRX function if the packet is not meant for it. However, if the destination address of the packet is that particular node then the receivePacket function is called. This function receives the packet with only the application data and no WifiMacHeader making the use of the data easy. Here we also separate the tag from the data and use the position and velocity data according to our use.

Whenever a node has to send a packet the application schedules a broadcast function to send the packets to the RSUs. Here we can set the priority of the data and customize the data rate and bandwidth of a packet.
Next we add the Mac48Address of all the nodes which are acting as RSUs. These addresses follow a sequential pattern wherein a node with node ID 00 will have the address as XX-XX-00:00:00:00:00:01. The format of the address is hexadecimal. Following this we create the various counters needed to maintain the logic of the code and store the historic position and velocity of the nodes to check the logic for sending the packet. Next, we add the position and velocity of the node in all three coordinates to the packet tag and mark the packet tag to be attached to the packet. Next, we start the logic for the filter in the packet and if a node passes the logic then it sends the packet to the RSUs. However, the WAVE net devices can only send the data over a length of 180 to 190 units on the map which roughly translates to 180-190 metres in the real life comparison so only the nearby RSUs receive the packet. Following this, the broadcast function is recursively called and waits for the broadcast interval and restarts the process for the next packet of the same node or the next node.

Once the application is set we start the simulation for the total simulation time which is the last exit time of the node from the .tcl file.

From our experiments, we have observed that a total of 4,48,200 packets are created by all the nodes during the whole simulation which are checked using our filtering policy. Out of these, only 1,30,032 packets are sent which is 29.01%. This shows a reduction of 70.98% in the data transfer by applying the filtering policy.

```
Sent: 3672 out of 18000
Sent due to velocity: 1908 out of 18000
Sent due to position: 1764 out of 18000
Payload (size=1000)
0x555562859a50 0x55556256fd70 35036 02-06-00:00:00:00:00:0b 02-06-00:00:00:00:00:1b
ReceivePacket() : Node 26 : Received a packet from 02-06-00:00:00:00:00:0b Size:1096
    From Node Id: 10 at 509.39:187.78:0 moving at 0:0:0 Packet Timestamp: +700088000.0ns delay=+320724325.0ns
0x555562b201a0
    Previous pos: 1260:0:0  Current pos: 1260:0:0
    DIfference: 0
```

Here we can see the nodes are receiving the packets from the sender node. In this example the packet is being sent even though the difference is zero since the last 5 packets were dropped. We can see the address of the sender and the receiver node in the above screenshot.

6. Result

In this simulation, we have kept the threshold to filter data on the smaller side. The threshold for velocity is small enough that it covers cases. The threshold for position is small enough that it covers cases where and cases where nodes are travelling with constant velocity. Such conditions with low thresholds make sure that there is a minimal loss of data. Even after that, we still see a major drop of about 71% in the number of packets sent. This shows how many of these packets are sent by nodes which are stationary and are not as significant. Such unnecessary packets use the network bandwidth and RSU processing power which could be otherwise freed to increase speed and number of clients. Hence, this filtering method can improve data transfer efficiency in VANETs.

7. Conclusion and Future work

This experiment shows that this filtering method with considerably strict conditions can be used to improve the data transfer between vehicles and RSUs with minimal loss of data. This study can be further improved by transferring data to a single nearest RSU instead of all the RSUs in the vicinity. Further, all the RSUs can be connected through a station using a different protocol for communication. Also, with higher computational power this simulation can be further expanded to include a bigger area, more vehicles and more RSUs. We can also include security measures like encrypting the data before sending it using homomorphic encryption.

8. References

1. NS3 documentation - https://www.nsnam.org/documentation/
2. NS3 google group - https://groups.google.com/g/ns-3-users
3. https://www.youtube.com/c/AdilAlsuhaim/featured
4. https://www.youtube.com/c/tspradeepkumar/featured
5. Stack Overflow for doubts https://stackoverflow.com/