Veins Application Documentation

Sakshi Roongta

July 2019

Contents

1 Overview

This project simulates a network between cars and an RSU (road side unit) at a four-way intersection. The cars send messages to the RSU with their current road ID, which direction they want to turn, and other necessary information. The RSU will figure out which vehicles are allowed to go and then send a message containing a list of vehicles allowed to go out to all the vehicles. If the vehicle is allowed to go, then it will proceed through the intersection. Otherwise, it will slow down and stop at the intersection and wait until it is allowed to go.

The project uses an open-source traffic simulator called SUMO, an event-based network simulator called OMNeT++, and a vehicle network simulator called Veins which is built on top of these two frameworks and connects them. Veins runs the two simulators in parallel, connected with a TCP socket. The movement of the cars in the SUMO simulation will correspond with the movement of nodes in the OMNeT++ simulation. This is all handled through a part of SUMO known as TraCI (Traffic Control Interface). TraCI gives access to a running SUMO simulation and allows you to change its behavior dynamically. The TraCI commands have been implemented in Veins. Veins is designed to be the basis of writing application-specific simulation code.

2 Installation

In order to run this project, you have to install SUMO, OMNeT++, and Veins. In order to set them up, I would recommend using this tutorial as a guideline: https://veins.car2x.org/tutorial/ from the Veins documentation page as that is what I used. Make sure that the versions of SUMO, OMNeT++, and Veins that you are using are compatible. I used SUMO 0.32.0, OMNeT++ 5.3, and Veins 4.7, as the tutorial recommended. I used a Windows laptop to download everything. The steps will be slightly different if you are using a Mac.
Steps:

1. **Installing Packages:**
   If you are on a Windows laptop, you do not need to download any additional packages. If you are running this on a Mac, you will need to likely install all these packages via Macports as so.

   ```
   sudo port install bison zlib tk blt libxml2 libtool xercesc3 proj
   gdal fox
   ```

2. **Downloading SUMO**

   (a) Use the source file on SourceForge
   https://sourceforge.net/projects/sumo/, click on Project Activity,

download the src zip file for version 0.32.0 and unpack it as `C:\Users\user\src\sumo-0.32.0`.

(b) On a Windows or Linux laptop, you should be able to directly run the executable `sumo.exe`. On a Mac, you will have to configure and make.

```
./configure --with-fox-includes=/usr/includes=/usr/include/fox-1.6 \
--with-gdal-includes=/usr/include/gdal --with-proj-libarires=/usr \
--with-gdal-libraries=/usr --with-proj-gdal

make
sudo make install
```

To make sure the Mac instructions worked, run `sumo-gui` from the terminal.

3. **Installing OMNeT++**

(a) Go to https://omnetpp.org/download/old and download version 5.3.

(b) On Windows, this should give you a script `C:\Users\user\src\omnetpp-5.3\mingwenv.cmd` that will open a MinGW command line window. Once you open that Window, build OMNet++ by running

```
./configure

make
```

If it worked, then you should be able to run `omnetpp` in the command line to launch the OMNeT++ IDE. ON Mac, you should just be able to unpack the file and directly open the IDE. Just use the default workspace which is `C:\Users\user\src\omnetpp-5.3\samples`.

4. **Installing Veins**

(a) Download Veins 4.7 from https://veins.car2x.org/download/

(b) In the OMNeT++ IDE, click `File > Import > General: Existing Projects into Workspace` and select the directory you unpacked the module framework to as the root directory and clicking finish.

(c) Build the newly imported project by choosing `Project > Build All`.

5. **Making Sure SUMO Works**

To make sure SUMO is working, open the OMNeT++ MinGW command line

window, change the current directory to `vein-4.7/examples/veins`, and then run `/c/Users/user/src/sumo-0.32.0/bin/sumo.exe -c` erlangen.sumo.cfg. You should see a a line saying "Loading configuration... done.", then a simulation time step counter running from 0 to 1000 and disappearing again.

6. **Running Veins Tutorial Demo**
   The last step is to make sure that everything has been installed correctly. In your OMNeT IDE, right click on `veins-4.7/examples/veins/omnetpp.ini` and choosing `Run As > OMNeT++ simulation`. Allow access to SUMO through any firewalls. You will need to run SUMO at the same time as OMNeT. In order to do so, in the OMNeT++ MinGW command line window, run

   `/c/Users/user/src/veins-4.7/sumo-launchd.py -vv -c /c/Users/user/src/sumo-0.32.0/bin/sumo.exe`

3 Running This App

1. Make sure you have Veins, OMNeT++, and SUMO installed as described in the previous section.

2. Fork and clone the Github repository https://github.com/sakshir0/V2V-Simulator-App to your personal computer.

3. Open the OMNet++ IDE using this project folder as your workspace.

4. In the OMNeT++ IDE, import the project by

   `File > Import > General: Existing Projects Into Workspace`

   and select the project.

5. Click the project folder and then click `Project > Build All`. If the project failed to build and gives you an error such as 'fatal error: veins/modules/ applications/ieee80211p/ BaseWaveApplLayer.h: No such file or directory', then you need to change the Makefile settings. Right click on the

   `Project folder > Properties > OMNeT++ > makemake > Options > Compile Tab`

   and include the path to the veins-4.7 src folder.

6. Right click on the omnetpp.ini file and click `Run As > OMNeT++ Simulation`. This file is the initialization file. Remember that you must run this command in the terminal

   ```
   /c/Users/user/src/veins-4.7/sumo-launchd.py -vv -c /c/Users/
   user/src/sumo-0.32.0/bin/sumo.exe
   ```

   in order to run the simulation.

7. When the simulation pops up, it should ask you what configuration you want to use. Select WithChannelSwitching as the configuration.

8. If you want to see the simulation run in the SUMO GUI, simply change sumo.exe to sumo-gui.exe in the command. If you do this, you must first click run in the OMNeT++ simulation GUI, then click run again in the SUMO simulation GUI.

4 Explanation of Code

4.1 SUMO

4.1.1 Defining the Intersection

The files that define the intersection in SUMO are:

- intersect.nod.xml - defines the nodes
- intersect.edg.xml - defines the edges
- intersect.typ.xml - defines types of edges and their characteristics
- intersect.con.xml - defines the connections

Using these files, I can generate the intersect.net.xml file that is the SUMO binary that defines the intersection.

Figure 1 provides an overview of the nodes, edges, and connections I used to define the intersection in SUMO. Each of the circles represents a node in SUMO. There are two types of edges: an "in" and an "out" edge. There are inner and outer edges between each of the outer nodes to each of the inner nodes and between each of of the inner nodes to node 0. The edges are connected by connections that define how vehicles can move from one edge to another.

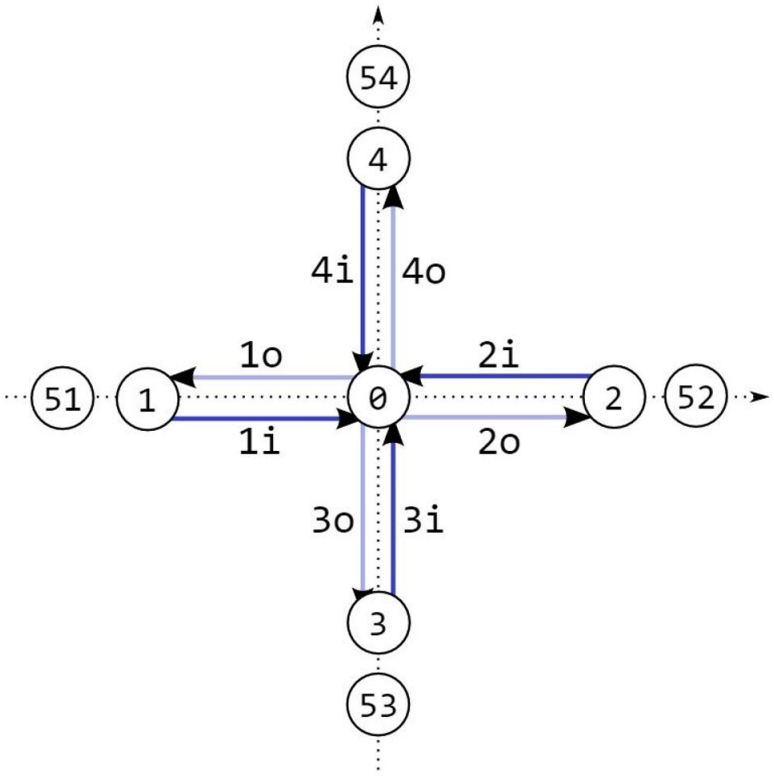

Figure 1: Overview of Intersection

Nodes are the reference points on a map for SUMO and edges connect these paths. You can define priority, the number lanes, the maximum speed, and other parameters of an edge. Connections allow vehicles to be routed from one edge to another. I defined connections for going straight, right turns, and left turns in SUMO from every "in" edge to a certain "out" edge. For example, a left turn from node 1 would be defined as a connection from "1i" to to "4o".

4.1.2 Defining the Cars and the Routes

The intersect.rou.xml file contains all the information about defining the cars and the routes.

There are three types of vehicles: fast, slow, and emergency. Each vehicle type has a defined acceleration and deceleration rate, max speed, and color in the simulation. You can define other parameters about the vehicles as well. For example, the emergency vehicle has a speed mode of 1.5, meaning that it can travel at 1.5x the maximum speed on the defined edge.

I defined all possible routes for the intersection and separated them by straight, right turns, and left turns. I also defined example vehicles for each of the routes. These vehicles are commented out but I left them in the code so that it is easy for someone to copy and paste the vehicles they want in the simulation. Currently, there are four actual vehicles in the simulation, defined under the vehicles section.

4.1.3 Configuration

The other xml files configure SUMO to work with Veins and OMNeT++. The antenna.xml file defines the influence of antenna characteristics and car panorama glass roofs on Car2Car Communication. This was taken from the Veins example project.

The config.xml file defines the type of analogue model and decider type. This was also taken from the Veins example project.

The intersect.netc.xml and intersect.launchd.xml allows Veins to find the correct files for each type and product a .net.xml file as output.

4.2 OMNeT++

4.2.1 Message Types

There are two types of messages in this application. IntersectMessages is the type of message the cars send to the RSU (road side unit). This message is defined in IntersectMessage.msg. It contains the vehicle ID, vehicle type, time sent, road ID, direction the car is trying to go, and if the car has passed the intersection yet or not.

The other type of message is an RSUMessage which the RSU sends to the cars. This message is defined in RSUMessage.msg and it contains a list of vehicleID's that are currently allowed to go through the intersection.

Both message types extend a type of message called a Wave Short Message which is the default type of message used in the Veins tutorial example. This allows me to use Veins functions to send the message down through the network layer and back up through the application layer. When you define a .msg file in OMNeT++, the compiler will automatically generate C++ code and a .h file to go with it. I use the functions generated to create and populate the message as well as to retrieve the fields of the message when it arrives.

4.2.2 Network

The OMNeT++ network used in this simulation is defined in IntersectionScenario.ned. It is a simple extension of the Veins Scenario network with an RSU added in. Figure 2 contains the Scenario network with explanations added in.

```
network Scenario
{
    parameters:
        double playgroundSizeX @unit(m); // x size of the area the nodes are in (in
        meters)
        double playgroundSizeY @unit(m); // y size of the area the nodes are in (in
        meters)
        double playgroundSizeZ @unit(m); // z size of the area the nodes are in (in
        meters)
        @display("bgb=$playgroundSizeX,$playgroundSizeY");
    submodules:
        obstacles: ObstacleControl { //models the obstacles for the radio signals
            @display("p=240,50");
        }
        annotations: AnnotationManager { //manages drawings of network traffic
            @display("p=260,50");
        }
        connectionManager: ConnectionManager { //manages wireless connections
            parameters:
                @display("p=150,0;i=abstract/multicast");
        }
        world: BaseWorldUtility { //defines a world object
            parameters:
                playgroundSizeX = playgroundSizeX;
                playgroundSizeY = playgroundSizeY;
                playgroundSizeZ = playgroundSizeZ;
                @display("p=30,0;i=misc/globe");
        }
        manager: TraCIScenarioManagerLaunchd { //manages connection to SUMO through
        TraCI
            parameters:
                @display("p=512,128");
        }

    connections allowunconnected: //allows cars to communicate in a mesh
}
```

Figure 2: Scenario Defined in Veins

When we use SUMO to add vehicles to the application, Veins uses a specific network called Car. This class will define the objects needed for the application, a network interface card (NIC), a mobility class, and connect the NIC to the application. Figure 3 contains the Car class with explanations added in.

```
module Car
{
    parameters:
        string applType; //type of the application layer
        string nicType = default("Nic80211p"); // type of network interface card
        string veinsmobilityType =
        default("org.car2x.veins.modules.mobility.traci.TraCIMobility"); //type of
        the mobility module
    gates:
        input veinsradioIn; // gate for sendDirect
    submodules:
        //this will handle communication
        appl: <applType> like org.car2x.veins.base.modules.IBaseApplLayer {
            parameters:
                @display("p=60,50");
        }
        //models wireless properties
        nic: <nicType> like org.car2x.veins.modules.nic.INic80211p {
            parameters:
                @display("p=60,166");
        }
        //manages mobility of car in Sumo through TraCI to make sure nodes in
        //OMNeT++ and SUMO car are synced
        veinsmobility: <veinsmobilityType> like
        org.car2x.veins.base.modules.IMobility {
            parameters:
                @display("p=130,172;i=block/cogwheel");
        }
    //connections from car layer to the network layer and vice versa
    connections:
        nic.upperLayerOut --> appl.lowerLayerIn;
        nic.upperLayerIn <-- appl.lowerLayerOut;
        nic.upperControlOut --> appl.lowerControlIn;
        nic.upperControlIn <-- appl.lowerControlOut;

        veinsradioIn --> nic.radioIn;
```

Figure 3: Car Module in Veins

4.2.3 Initialization

The omnetpp.ini file contains all the parameters for initializing the OMNeT++ simulation. Some important ones include the simulation time limit, the position of the RSU, and the configuration and whether or not the application allows beaconing and channel switching. When I run the application, I always use

4.3 Application Layer

The application is defined using a .ned, .h, and .cc file called IntersectionApp. The ned file defines the module application and registers the class with it. It simply uses the Base Wave Application Layer for Veins.

The header file contains all the function definitions used in the application. The class takes from the Basic Wave Application Layer in Veins that allows you to build an application on top of the network layer. Figure 4 has snippets of that class definition. The class is split into RSU code, vehicle code, and general code. We initialize the isRSU boolean variable which keeps track of if we are a car or an RSU. There are function headers in the C++ file that go into more depth into what exactly each function does. The initialize, IntersectApp, handleSelfMsg and handleLowerMsg are functions that are redefined from the Base Wave Application Layer. The handlePositionUpdate function is called every time the object moves which is almost constantly and is also redefined from the previous class. The way messages are being delivered is through a function called sendDown from Veins which sends a message down to the network layer with a specified delay. This function will eventually trigger the handleLowerMsg function which, depending on the type of message, will call the appropriate message handler (either onISM or onRSM).

On a high level, the flow of control in this app is as follows:

- initialize function initializes everything
- handlePositionUpdate is called as car will move
- That function has the cars send a message to the RSU
- The handleSelfMsg and handleLowerMsg will receive the message
- The onISM function will be called
- That function has the RSU send a message to the cars
- The handleSelfMsg and handleLowerMsg will receive the message
- The onRSM function will be called

- If a car is allowed to go through the intersection, it will set the bool canGo to True
- If this bool is true, then in the handlePositionUpdate function, it will be allowed to go through intersection

```
class IntersectionApp : public BaseWaveApplLayer {
    public:
        virtual void initialize(int stage);
    protected:
        ~IntersectionApp();
        //RSU
        simtime_t delayTimeRSU = 0;
        simtime_t lastSentRSU; //last time RSU sent msg
        std::vector<IntersectMessage*> RSUData;
        virtual void onISM(IntersectMessage* wsm);
        virtual void addData(IntersectMessage* ism);
        virtual void removeData(IntersectMessage* ism);
        std::vector<IntersectMessage*> yieldToRight(std::vector<IntersectMessage*> vehicles);
        std::vector<IntersectMessage*> yieldToRightLeftTurns(std::vector<IntersectMessage*> vehicles);
        std::list<const char*> getVehicleIds(std::vector<IntersectMessage*> vehicles);
        std::list<const char*> priorityCars(std::vector<IntersectMessage*> vehicles);
        std::list<const char*> calculateAllowedVehicles();

        //Vehicles
        simtime_t delayTimeCars = 0;
        simtime_t lastSent; // the last time car sent a message
        bool canGo; //if the car can go through intersection
        virtual void onRSM(RSUMessage *rsm);
        int getDirection(std::string currRoad, std::string nextRoad);
        void populateISM(IntersectMessage *ism, bool passed);
        virtual void handlePositionUpdate(cObject* obj);

        //General
        bool isRSU; //if it is an RSU or a car
        virtual void handleSelfMsg(cMessage* msg);
        virtual void handleLowerMsg(cMessage *msg);
};
```

Figure 4: Intersect Application Header File