SUMO Veins tutorial

Stahorszki, Peter Bence

2020 - May

Contents

1 About Veins

Veins, the Open Source vehicular network simulation framework, ships as a suite of simulation models for vehicular networking. These models are executed by an event-based network simulator (OMNeT++) while interacting with a road traffic simulator (SUMO). Other components of Veins take care of setting up, running, and monitoring the simulation.

This consitutes a simulation framework. What this means is that Veins is meant to serve as the basis for writing application-specific simulation code. While it can be used unmodified, with only a few parameters tweaked for a specific use case, it is designed to serve as an execution environment for user written code. Typically, this user written code will be an application that is to be evaluated by means of a simulation. The framework takes care of the rest: modeling lower protocol layers and node mobility, taking care of setting up the simulation, ensuring its proper execution, and collecting results during and after the simulation.

Veins contains a large number of simulation models that are applicable to vehicular network simulation in general. Not all of them are needed for every simulation – and, in fact, for some of them it only makes sense to instantiate at most one in any given simulation. The simulation models of Veins serve as a toolbox: much of what is needed to build a comprehensive, highly detailed simulation of a vehicular network is already there. Still, a researcher assembling a simulation is expected to know which of the available models to use for which job. To give a trivial example, one would not want to use a path loss model designed for cities to simulate a freeway scenario.

Veins is an Open Source vehicular network simulation framework. What this means is that it (and all of its simulation models) are freely available for download, for study, and for use. Nothing about its operation is (or needs to be) kept secret. Any simulation performed with Veins can be shared with interested colleagues – not just the results, but the complete tool chain required for an interested colleague to reproduce the same results, to verify how they were derived, and to build upon the research performed.

Road traffic simulation is performed by SUMO, which is well-established in the domain of traffic engineering. Network simulation is performed by OMNeT++ along with the physical layer modelling toolkit MiXiM, which makes it possible to employ

accurate models for radio interference, as well as shadowing by static and moving obstacles.

Both simulators are bi-directionally coupled and simulations are performed online. This way, the influence of vehicular networks on road traffic can be modeled and complex interactions between both domains examined.

Domain specific models for vehicular networking build on this basis to provide a comprehensive framework that is still easy to learn and use.

For mor information, refer to the website.

2 Setup and first steps

2.1 Instant Veins

If you just want to try out Veins and don't want to spend the time to configure it on your machine, there is a preconfigured Linux instance which comes with Veins installed, all you have to do is download it, import it in Oracle VM VirtualBox, and start the machine. Download instant Veins and Oracle VM VirtualBox. I advice an older build, like 5.2.32, that is the recommended on the Veins website, and I could not get it to work with newer (6.1) release.

I recommend downloading a previous release of Instant Veins, 4.7.1, because I had problems with version 5.0.1, it broke everytime when I tried to move the VirtualBox window, or tried to change its resolution, and also, 4.7.1 was overall faster then 5.0.1.

After you downloaded Veins and installed VirtualBox, import and start the machine, and continue with the section about running Veins on Linux.

2.2 Windows

This tutorial will asume that you put your Veins related files into **C:/Users/user/src** folder, where **user** is your Windows username. Of course you can put your files in other locations, just change the paths given in this tutorial accordingly.

2.2.1 Download and install SUMO

Download the SUMO binaries (the zip file) and unpack them as

```
C:/Users/user/src/sumo-1.2.0
```

This should give you an executable

```
C:/Users/user/src/sumo-1.2.0/bin/sumo.exe
```

I tried SUMO versions 1.2.0 and 1.6.0, both worked flawlessly.

2.2.2 Download and build OMNeT++ 5

Download OMNeT++ for Windows and unpack it as

```
C:/Users/user/src/omnetpp-5.5.1
```

After extracted there should be a script:

```
C:/Users/user/src/omnetpp-5.5.1/mingwenv.cmd
```

After opening this script, we can start building OMNeT++ 5 by typing **./configure** in the opened console window. After the first command finished, type the **make** command. Afther it finished, you can use the omnetpp command to launch the OMNeT++ 5 IDE.

2.2.3 Download and build Veins

Download Veins and unpack it as:

```
C:/Users/user/src/veins-5.0
```

Import the project into your OMNeT++ IDE workspace by clicking **File > Import > General: Existing Projects into Workspace** and selecting the directory you unpacked the module framework to.

Build the newly imported project by choosing Project > Build All in the OMNeT++ 5 IDE. After the project built, you are ready to run your first IVC evaluations, but to ease debugging, the next step will ensure that SUMO works as it should.

2.3 Make sure SUMO is working

Add SUMO folder to the executables path, so you don't have to type the full path every time you want to run SUMO:

```
export PATH=$PATH:/c/Users/user/src/sumo-1.2.0/bin
```

This will only affect the mingw commandline window. In order to add sumo to your windows command line executable path, open **Environment variables** from the Start menu, and add the above mentioned path to the **Path** variable.

In the OMNeT++ MinGW command line window, you should be able to have SUMO simulate an example scenario by changing the current directory to:

```
cd C:/Users/user/src/veins-5.0/examples/veins/
```

and running:

```
sumo -c erlangen.sumo.cfg
```

to start SUMO. You should see a line saying **"Loading configuration... done."**, then - after a short while - with no further output be returned to the command line.

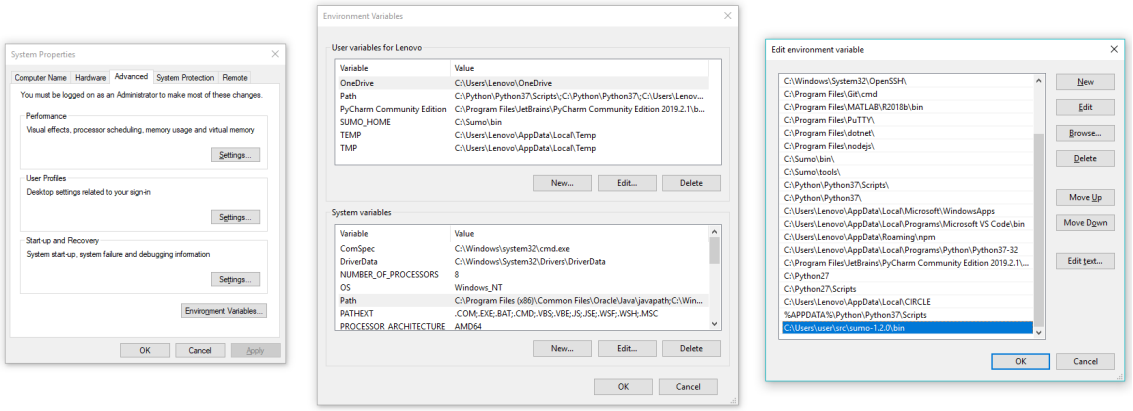

Figure 1: Windows edit environmental variables

To get an impression of what the example scenario looks like, you can also run it using **sumo-gui.exe**, but this is not required for Veins to work.

2.4 Final step: Run the Veins demo scenario

To save you the trouble of manually running SUMO prior to every OMNeT++ simulation, the Veins module framework comes with a small python script to do that for you. In the OMNeT++ MinGW command line window, move into the folder:

```
cd C:/Users/user/src/veins-5.0
```

and start it by running:

```
sumo-launchd.py -vv -c sumo.exe
```

This script will proxy TCP connections between OMNeT++ and SUMO, starting a new copy of the SUMO simulation for every OMNeT++ simulation connecting. The script will will print **Listening on port 9999** and wait for the simulation to start. Leave this window open and switch back to the OMNeT++ 5 IDE.

In the OMNeT++ 5 IDE, simulate the Veins demo scenario by right-clicking on **veins-5.0/examples/veins/omnetpp.ini** and choosing **Run As > OMNeT++** simulation. Don't forget to allow access to SUMO through any personal firewall you might run. Similar to the last example, this should create and start a launch configuration. You can later re-launch this configuration by clicking the green Run button in the OMNeT++ 5 IDE.

If everything worked as intended this will give you a working simulation scenario using OMNeT++ and SUMO running in parallel to simulate a stream of vehicles that gets interrupted by an accident.

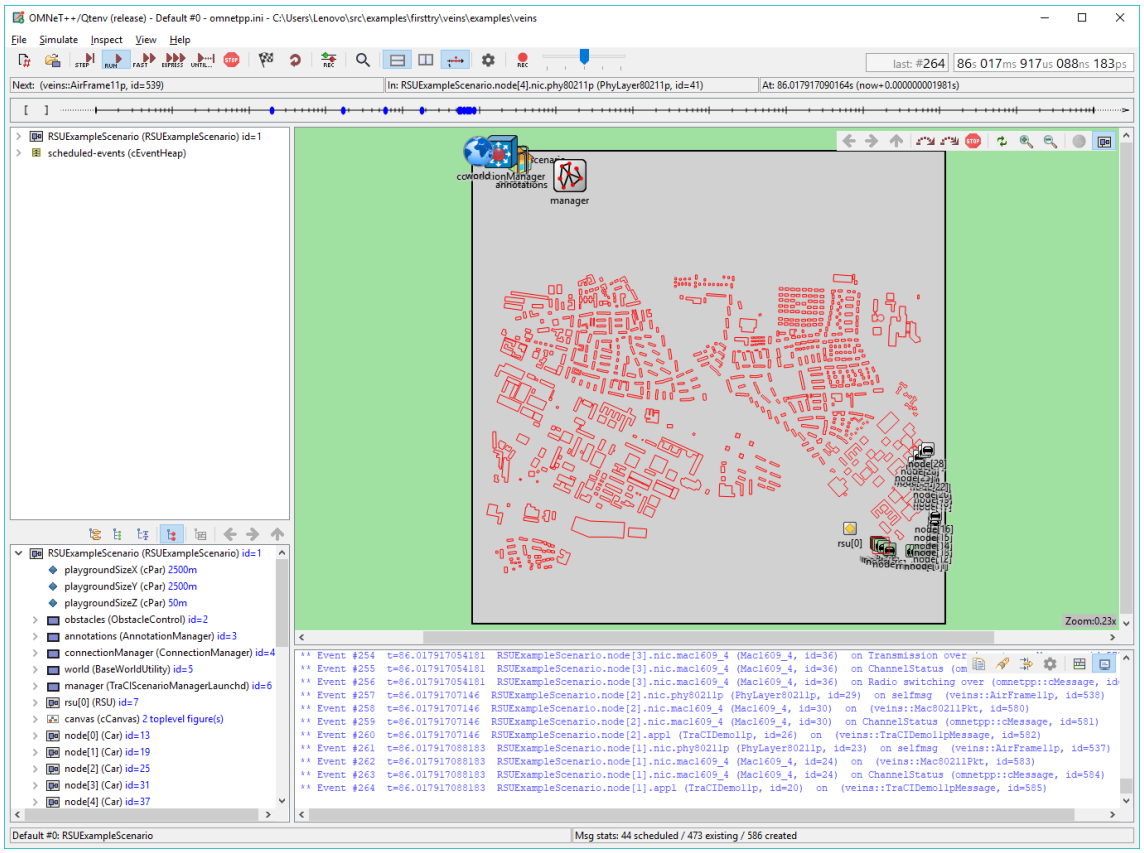

Figure 2: Running simulation

If you ran a debug build of the simulation (e.g., by choosing Debug as instead of Run as when launching the simulation by right-clicking on **omnetpp.ini** and acknowledging the prompt to build the simulation to debug mode), you will see a wealth of debug output in the log window. Note, however, that a simulation running in debug mode executes at lower speed.

Note: the simulation can run into problems, if you have multiple versions of Python installed, so try running the **sumo-launchd.py** with a specific version, or delete one of the Pyton version from your computer. I'm using **Python 3.7.4** in this tutorial.

2.5 Linux

In this section I will show you how to install OMNeT++, Veins and SUMO on Ubuntu 20.04.

2.5.1 Installing OMNeT++

Although the OMNeT install guide recommends Ubuntu 16.04 or 18.04, I could install it on 20.04 without any issues, I just had to make a few adjustments.

I had to expand the repository sources list. To do this, open the sources file:

```
sudo nano /etc/apt/sources.list
```

And add the following repositories at the end of the file. After you are done with the installation, you can delete these.

```
###### Ubuntu Trusty Repos
deb http://hu.archive.ubuntu.com/ubuntu/ trusty main restricted ...
    universe multiverse
deb-src http://hu.archive.ubuntu.com/ubuntu/ trusty main restricted ...
    universe multiverse

###### Ubuntu Xenial Repos
deb http://hu.archive.ubuntu.com/ubuntu/ xenial main restricted ...
    universe multiverse
deb-src http://hu.archive.ubuntu.com/ubuntu/ xenial main restricted ...
    universe multiverse

###### Ubuntu Bionic Repos
deb http://hu.archive.ubuntu.com/ubuntu/ bionic main restricted ...
    universe multiverse
```

```
deb-src http://hu.archive.ubuntu.com/ubuntu/ bionic main restricted ...
    universe multiverse
```

After this run:

```
$ sudo apt-get update
$ sudo apt-get upgrade
```

When thats finished, install the packages:

```
$ sudo apt-get install build-essential gcc g++ bison flex perl
python python3 qt5-default libqt5opengl5-dev tcl-dev tk-dev ...
    libxml2-dev zlib1g-dev default-jre doxygen graphviz ...
    libwebkitgtk-3.0-0
```

To add Qtenv with 3D visualization support, follow these steps:

```
$ sudo add-apt-repository ppa:ubuntugis/ppa
$ sudo apt-get update
# install osgearth development package (and OpenSceneGraph, too)
$ sudo apt-get install openscenegraph-plugin-osgearth libosgearth-dev
```

To enable the optional parallel simulation support you will need to install the MPI packages:

```
$ sudo apt-get install openmpi-bin libopenmpi-dev
```

The optional Pcap library allows simulation models to capture and transmit network packets bypassing the operating system's protocol stack. It is not used directly by OMNeT++, but models may need it to support network emulation.

```
$ sudo apt-get install libpcap-dev
```

Now download OMNeT++. Move it to a directory, for example:

```
/home/<you>/veins_framework/src/
```

It is important, to move the following components - or at least Veins - in a folder called **.../../src**, because when we will want to add the Veins project to OMNeT++, it will only be recognized az a project, if it is in a **src** directory. Then extract it:

```
$ tar xvfz omnetpp-5.6.1-src.tgz
```

OMNeT++ needs its bin/ directory to be in the path. To add bin/ to PATH

temporarily (in the current shell only), change into the OMNeT++ directory and source the setenv script:

```
$ cd omnetpp-5.6.1
$ . setenv
```

Add OMNeT++ to your environment variables with a text editor:

```
$ nano ¬/.bashrc
```

```
export PATH=$HOME/veins_framework/src/omnetpp-5.6.1/bin:$PATH
```

In the top-level OMNeT++ directory, type:

```
$ ./configure
```

The configure script detects installed software and configuration of your system. It writes the results into the Makefile.inc file, which will be read by the makefiles during the build process.

When ./configure has finished, you can compile OMNeT++. Type in the terminal:

```
$ make
```

Finally, add the OMNeT++ libs to the libraries path, this is needed in order to be able to build the Veins project we will import in a later step:

```
$ echo $LD_LIBRARY_PATH
```

If nothing is displayed, add a default path value, and append our OMNeT++ libraries:

```
$ LD_LIBRARY_PATH=/usr/local/lib
$ LD_LIBRARY_PATH= ...
    $LD_LIBRARY_PATH:/home/<you>/veins_framework/src/omnetpp-5.6.1/lib/
```

After finished with everything, you should have yourself a working OMNeT++ instance. Type **omnetpp** in your terminal window, and start the IDE, to verify the installation.

Note: would you run into any trouble, refer to the official installation guide, for further assistance.

2.5.2 Installing SUMO

Install the required tools and libraries:

```
$ sudo apt-get install cmake python g++ libxerces-c-dev ...
    libfox-1.6-dev libgdal-dev libproj-dev libgl2ps-dev swig
```

Move to the directory you chose for your veins framework and get the source code (you may need to install **git** for this):

```
$ git clone --recursive https://github.com/eclipse/sumo
```

Add SUMO_HOME path:

```
$ export SUMO_HOME="$PWD/sumo"
```

Build SUMO:

```
$ mkdir sumo/build/cmake-build && cd sumo/build/cmake-build
$ cmake ../..
$ make -j$(nproc)
```

Finally, add the SUMO bin folder to the executables path, this way you don't have to type in the full path to run it:

```
$ nano ¬/.bashrc
```

```
export PATH=$HOME/veins_framework/src/sumo/bin:$PATH
```

2.5.3 Download and build Veins

Download Veins and unpack it as:

```
/home/<you>/veins_framework/src/veins-5.0
```

Import the project into your OMNeT++ IDE workspace by clicking **File > Import > General: Existing Projects into Workspace** and selecting the directory you unpacked the module framework to.

Build the newly imported project by choosing Project > Build All in the OMNeT++ 5 IDE. After the project built, you are ready to run your first IVC evaluations, but to ease debugging, the next step will ensure that SUMO works as it should.

2.5.4 Finally: run the Veins demo scenario

To save you the trouble of manually running SUMO prior to every OMNeT++ simulation, the Veins module framework comes with a small python script to do that for you. In the OMNeT++ MinGW command line window, move into the folder:

```
cd /home/<you>/veins_framework/src/veins-5.0
```

and start it by running:

```
sumo-launchd.py -vv -c sumo
```

This script will proxy TCP connections between OMNeT++ and SUMO, starting a new copy of the SUMO simulation for every OMNeT++ simulation connecting. The script will will print **Listening on port 9999** and wait for the simulation to start. Leave this window open and switch back to the OMNeT++ 5 IDE.

In the OMNeT++ 5 IDE, simulate the Veins demo scenario by right-clicking on **veins-5.0/examples/veins/omnetpp.ini** and choosing **Run As > OMNeT++** simulation. Don't forget to allow access to SUMO through any personal firewall you might run. Similar to the last example, this should create and start a launch configuration. You can later re-launch this configuration by clicking the green Run button in the OMNeT++ 5 IDE.

If everything worked as intended this will give you a working simulation scenario using OMNeT++ and SUMO running in parallel to simulate a stream of vehicles that gets interrupted by an accident.

If you ran a debug build of the simulation (e.g., by choosing Debug as instead of Run as when launching the simulation by right-clicking on **omnetpp.ini** and acknowledging the prompt to build the simulation to debug mode), you will see a wealth of debug output in the log window. Note, however, that a simulation running in debug mode executes at lower speed.

Note: the simulation can run into problems, if you have multiple versions of Python installed, so try running the **sumo-launchd.py** with a specific version, or delete one of the Pyton version from your computer. I'm using **Python 3.7.4** in this tutorial.

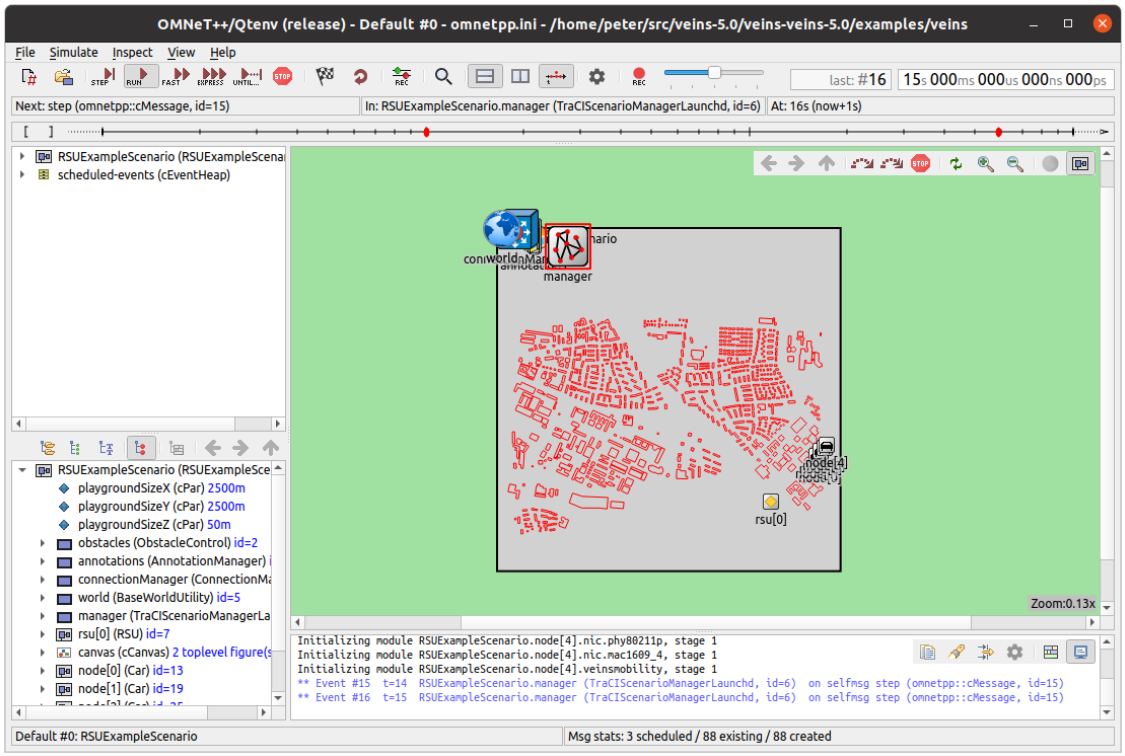

Figure 3: Running simulation

3 Example project

The recommended way to proceed is to simply modify the tutorial simulation to get started with using Veins, but you might want to create a dedicated OMNeT++ project. The easiest way is using cookiecutter-veins-project, which is a Cookiecutter template. First we need to install Cookiecutter. Open a command line window, and type:

```
pip install --user cookiecutter
```

After the installation is done, make sure that the Python scripst are executable from command line. Add **%APPDATA%/Python/Python3x/Scripts** to the environment variables, where Python3x should correspond to your installed Python version. Then execute:

```
cookiecutter gh:veins/cookiecutter-veins-project
```

This will download the Cookiecutter template and ask you some questions, e.g., the project name and which additional module libraries you want the project to use (for which you can safetly select the default answer by pressing the Enter key). It will then download the current version of Veins and create a new OMNeT++ project that is using Veins.

3.1 Explaining Cookiecutter project

In the next part of this tutorial I will explain to you the inner workings of the Cookiecutter VEINS project, to help you get started on customizing and building your own projects. You will get to know how to implement sending messages to vehicles in your simulation based on specific conditions and how to handle these messages.

Note: this tutorial assumes that you have some knowledge of SUMO and TraCI, and also you familiarized yourself with the basics of OMNeT++.

3.1.1 Initial steps, introduction to simulation

This demo simulation shows a traffic situation, where one vehicle suffers and accident and stays in that place for a given amount of time. When the accident happens, the vehicle sends out a message to the road side unit (RSU) and the other vehicles. The RSU repeats the message with a bit of delay, so vehicles which arrived after the first wave of messages would know about the accident too. When a vehicles

receive the message, it gets a chance to change routes, after that they also replay the received message.

3.1.2 Configuration

The first step we take is open omnetpp.ini in the IDE. Here we will see an overwhelming amount of parameters, but we will only focus on the essentials.

```
//...
    //the location of the project entry point file
    ned-path = .
//...
    //the location of the simulation scenarion NED file
    network = RSUExampleScenario
//...
    ###########################################################
    #            Simulation parameters                        #
    ###########################################################
    //allow debug on errors, when error occurs, switches to debug mode and jump
    ↪  to the error location
    debug-on-errors = true
    //duration of the simulation
    sim-time-limit = 200s
//...
    //size of the network the nodes takes place in
    *.playgroundSizeX = 2500m
    *.playgroundSizeY = 2500m
    *.playgroundSizeZ = 50m
//...
    ###########################################################
    #            TraCIScenarioManager parameters              #
    ###########################################################
    //location of the sumo launch file for TraCIScenarioManagerLaunchd
    *.manager.launchConfig = xmldoc("erlangen.launchd.xml")
//...
    ###########################################################
    #                       RSU SETTINGS                      #
    #                                                         #
    #                                                         #
    ###########################################################
    //position of the RSU
    *.rsu[0].mobility.x = 2000
    *.rsu[0].mobility.y = 2000
    *.rsu[0].mobility.z = 3
    //application layer for RSU
    *.rsu[*].applType = "TraCIDemoRSU11p"
```

```
//...
    ##########################################################
    #                       App Layer                        #
    ##########################################################
    //application layer for vehicles
    *.node[*].applType = "TraCIDemo11p"
//...
    ##########################################################
    #                       Mobility                         #
    ##########################################################
    //starting position, number of accidents, time and duration of accident
    *.node[*].veinsmobility.x = 0
    *.node[*].veinsmobility.y = 0
    *.node[*].veinsmobility.z = 0
    *.node[*].veinsmobility.setHostSpeed = false
    *.node[*0].veinsmobility.accidentCount = 1
    *.node[*0].veinsmobility.accidentStart = 73s
    *.node[*0].veinsmobility.accidentDuration = 50s
//...
```

3.1.3 Nodes

The participants in the V2x communications are represented as nodes. We use two types of NED nodes, the car and the RSU.

veins.nodes.Car: in this NED file we define the vehicle in the simulation. We set up a mobility type for the vehicle, which basically allows us to control the vehicle, and get informations from it through TraCI. The type of network and application layer defines the way the cars will communicate and act in the simulation.

```
//...
    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:
            appl: <applType> like org.car2x.veins.base.modules.IBaseApplLayer {
                parameters:
                    @display("p=60,50");
```

```
        }

        nic: <nicType> like org.car2x.veins.modules.nic.INic80211p {
            parameters:
                @display("p=60,166");
        }

        veinsmobility: <veinsmobilityType> like
        ↪  org.car2x.veins.base.modules.IMobility {
            parameters:
                @display("p=130,172;i=block/cogwheel");
        }

    connections:
        nic.upperLayerOut --> appl.lowerLayerIn;
        nic.upperLayerIn <-- appl.lowerLayerOut;
        nic.upperControlOut --> appl.lowerControlIn;
        nic.upperControlIn <-- appl.lowerControlOut;

        veinsradioIn --> nic.radioIn;

}
```

veins.nodes.RSU: the RSU node is almost exactly the same as the car module, with the exception that it does not define a separate mobility type.

3.1.4 Backround logic, messages

Mobility

veins.modules.mobility.traci.TraCIMobility: in this file the most important function we need to understand is **handleSelfMessage(cMessage* msg)**, and the usage of **scheduleAt(simtime_t t, cMessage *msg)** function. The others are mostly utility functions, that we would not want to change, for now at least.

```
//...
    void TraCIMobility::handleSelfMsg(cMessage* msg)
    {
        if (msg == startAccidentMsg) {
            getVehicleCommandInterface()->setSpeed(0);
            simtime_t accidentDuration = par("accidentDuration");
            scheduleAt(simTime() + accidentDuration, stopAccidentMsg);
            accidentCount--;
        }
        else if (msg == stopAccidentMsg) {
            getVehicleCommandInterface()->setSpeed(-1);
```

```
            if (accidentCount > 0) {
                simtime_t accidentInterval = par("accidentInterval");
                scheduleAt(simTime() + accidentInterval, startAccidentMsg);
            }
        }
    }
//...
```

With the **scheduleAt** function we can schedule the sending of a message. After the function was called, and the specified time passed, the **handleSelfMsg** function gets the message, and starts to process it. In this example it has two paths of execution, if the received message is **startAccidentMsg** or if it is **stopAccidentMsg**. In the first case, it sets the speed of the vehicle to 0, schedules a **stopAccidentMsg** message. When the **stopAccidentMsg** is received, the vehicle is started again, and the handler schedules another accident. The first message in this example originates from the **initialize** function.

Of course we could implement any kind of elaborate logic here, within the capabilities of the TraCI C++ interface, but for simplicity, we leave it as is.

Network layer

veins.modules.nic.Nic80211p: defines the Nic80211p network interface. We will not modify these parts of the example for now.

Application layer

veins.modules.application.ieee80211p.DemoBaseApplLayer: both application layer implementations in this example extend the **DemoBaseApplLayer** class. The functions here we are intrested in are the **handleLowerMsg(cMessage* msg)**, **sendDown(cMessage* msg)** and the **sendDelayedDown(cMessage* msg)**. Messeges are sent through the **sendDown** and **sendDelayedDown** functions. The received messages arrive in the **handleLowerMsg** function, where based on their types, are handed down to different functions. There are three types of V2x messages used in this example, **BSM - Basic Safety Message**, **WSA - Wave Service Announcement** and **WSM - Wave Short Message**, and each have their own handler function, **onBSM**, **onWSA** and **onWSM**.

```
//...
    void DemoBaseApplLayer::handleLowerMsg(cMessage* msg)
    {
```

```
        BaseFrame1609_4* wsm = dynamic_cast<BaseFrame1609_4*>(msg);
        ASSERT(wsm);

        if (DemoSafetyMessage* bsm = dynamic_cast<DemoSafetyMessage*>(wsm)) {
            receivedBSMs++;
            onBSM(bsm);
        }
        else if (DemoServiceAdvertisment* wsa =
        ↪  dynamic_cast<DemoServiceAdvertisment*>(wsm)) {
            receivedWSAs++;
            onWSA(wsa);
        }
        else {
            receivedWSMs++;
            onWSM(wsm);
        }

        delete (msg);
    }
//...
```

veins.modules.mobility.traci.TraCIDemo11p: this is the application layer, which is shared between all of the car nodes. Its parent class is **DemoBaseApplLayer**. The **onWSM** and **onWSA** functions are implemented here, the **onWSM** is the more interesting for us now. Once the vehicle receives the message, it will change route if possible, and repeat the received message after some time.

```
//...
    void TraCIDemo11p::onWSM(BaseFrame1609_4* frame)
    {
        TraCIDemo11pMessage* wsm = check_and_cast<TraCIDemo11pMessage*>(frame);

        findHost()->getDisplayString().setTagArg("i", 1, "green");

        if (mobility->getRoadId()[0] != ':')
        ↪  traciVehicle->changeRoute(wsm->getDemoData(), 9999);
        if (!sentMessage) {
            sentMessage = true;
            //repeat the received traffic update once in 2 seconds plus some
            ↪  random delay
            wsm->setSenderAddress(myId);
            wsm->setSerial(3);
            scheduleAt(simTime() + 2 + uniform(0.01, 0.2), wsm->dup());
        }
    }
//...
```

veins.modules.mobility.traci.TraCIDemoRSU11p: this is the application layer for the RSU node, which extends from the same parent class as **TraCIDemo11p**. There is less functions implemented here, we get the same **onWSM** function, just with a different implementation. When the RSU gets the message, and replays it with a few seconds delay.

```
//...
    void TraCIDemoRSU11p::onWSM(BaseFrame1609_4* frame)
    {
        TraCIDemo11pMessage* wsm = check_and_cast<TraCIDemo11pMessage*>(frame);

        //this rsu repeats the received traffic update in 2 seconds plus some
        ↪  random delay
        sendDelayedDown(wsm->dup(), 2 + uniform(0.01, 0.2));
    }
```

3.1.5 Running the example

I advise you to make some changes to the simulation, because it has way too many vehicles to be able to easily comprehend, what is happening on the screen.

I changed the flow in erlangen.rou.xml in the SUMO files, so it spawns less vehicles, with bigger timegaps between:

```
//...
    <flow id="flow0" type="vtype0" route="route0" begin="0" ...
        period="20" number="5"/>
//...
```

When you start the simulation, you will see some cars starting a journey, when suddenly the first one stops, and sends out messages to the RSU, and the other vehicles.

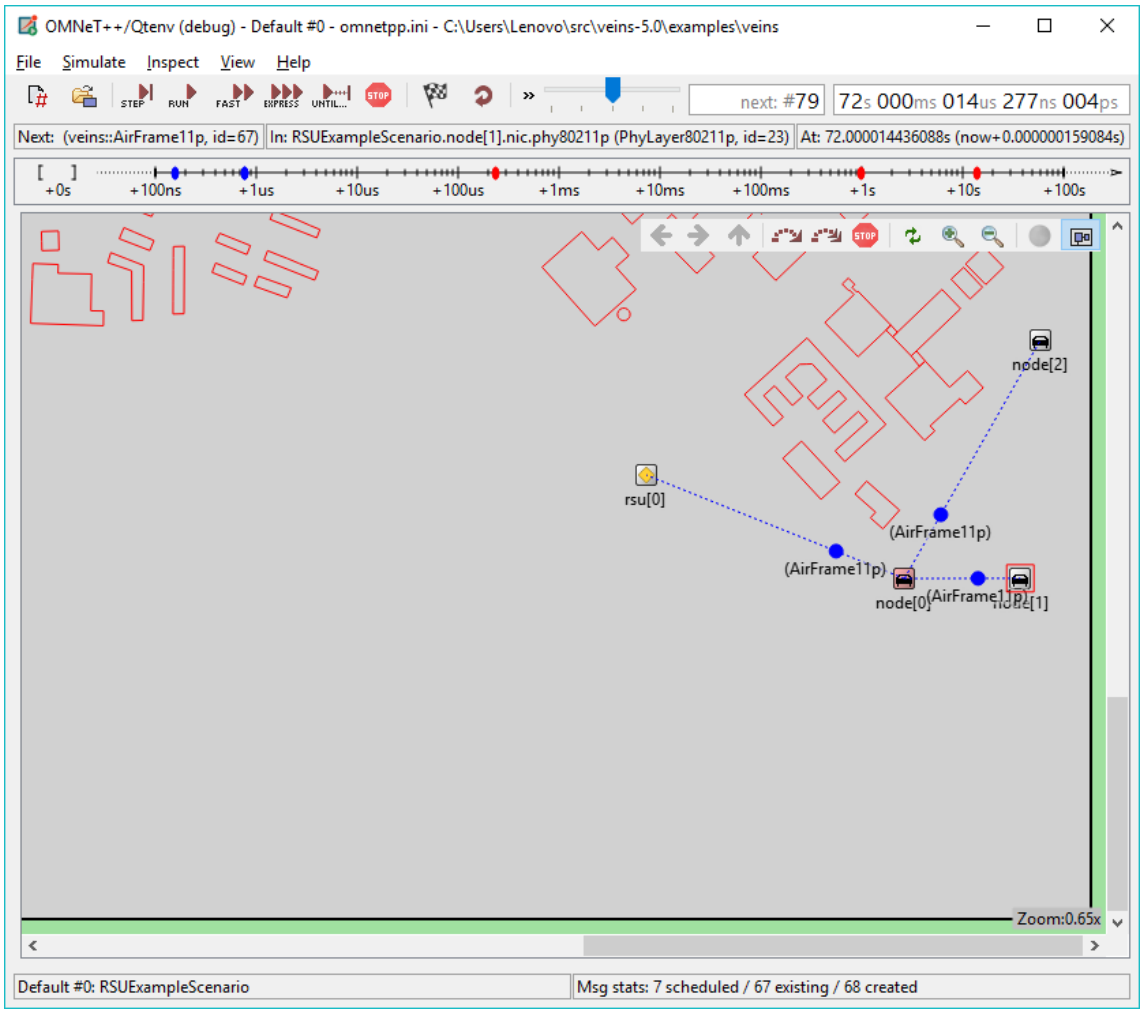

Figure 4: Messages sent after accident

After the participants received the message, they replay it, some vehicles will change route if it can. If you run the simulation in debug mode, and place breakpoints in the functions I mentioned above, you can follow the flow of the messages.