Table of Contents

1. Introduction

This report is meant to outline my effort in building a VANET simulation. I had decided to work in NS-3 which posed a considerable about of problems given it is still reasonable new. The available packages including the PhySim had build errors that I could be resolve so this project utilizes the standard YANS channel model.

This report may have missed some critical parts of research on the topic but this was only meant as a first attempt to gather as much of a broad-understanding on VANETs in the context of ITS applications.

2. Background

In this section I will introduce the topic of *vehicular ad-hoc networks* (VANETS) and my review of the past research that had been done in the topic that relates to this project.

VANETS vs. Typical Ad-hoc networks

VANETS, or Vehicle-2-Vehicle (V2V) networks as they are also known, are the extreme type of ad-hoc network which is characterized by high-mobility and high channel load. Given their critical role in Integrated Transportation Systems (ITS), they are also characterized by their real-time nature.

Current applications in ITS, include electronic tolling, navigation, and road safety. In the context of transportation safety, vehicle communication has to be more realizable then a best-effort network.

Also it should be mentioned that VANETS do have some positive characteristics that are different from regular ad-hoc networks. Since the mobile nodes are vehicles they in almost every situation will have semi-predicable movement, ie. Cars stay on the road. Also given that the actual communication equipment, also called on-board units (OBUs) in VANET literature, is powered by the car this means that power constraints are not as much an issue as for typical cellular hand-held equipment.

VANETS Specifications

Fully autonomous VANETS are still not a complete reality. The IEEE 802.11p amendment to the 802.11 standard is only a draft standard. Complementary with the IEEE 802.11p is the IEEE 1609. The latter outlines the specifications and implementation of the Upper layer protocol stack, while the former deals only with the physical layer and lower part of the MAC layer.

I have included these standards in the reference section.

The following referenced figure shows the relationship.

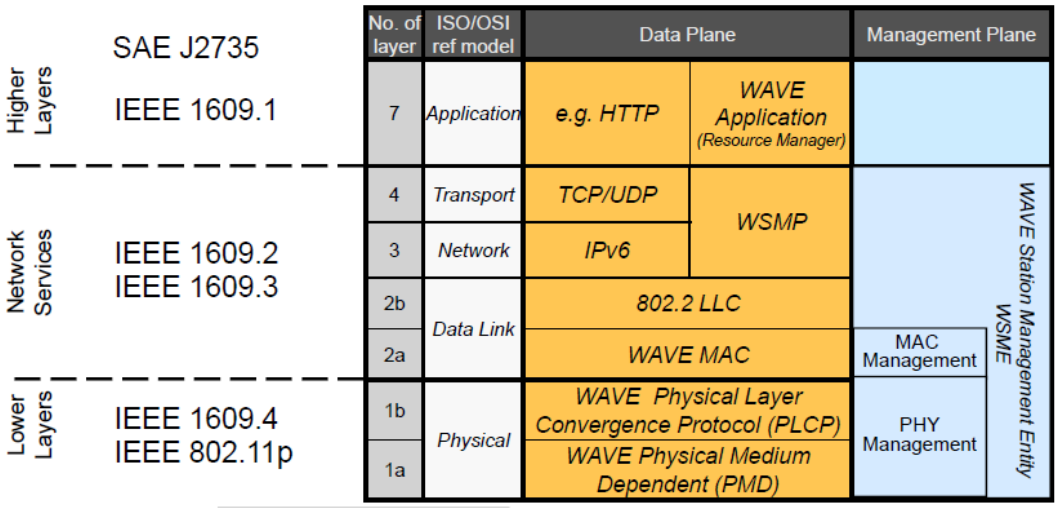


Figure: *The VANET communication protocol stack. Reference [3]*

This standard is also known as the dedicated short-range communication DSRC which also is the name for the frequency band 5.850-5.925 GHz. This band is further divided into 7 channels in which there is one control channel used for vehicle broadcasting.

Many of the reference I have seen give different specification as to data-rate and communication range. On average I would say that the typical range of DSRC communication is meant to be 300m – 500m and a data-rate of 3 – 20 Mbps (bits per sec).

The current equivalent European standard is the ETSI ES 202 663.

3. Scope of Project

VANETs are a broad topic and there is considerable amount of research effort devoted to them. This project is only meant to explore the Physical layer and MAC layers using the ns-3 simulation package.

As stated in reference [6] there exists a more realistic PHY and MAC implementation then the standard **YansWifi** model in the ns-3 package. This model is called the **PhySimWifi** and is available at

http://dsn.tm.uni-karlsruhe.de/english/ns3-physim.php

A realistic channel model is important when fast-fading affects the transmission success rate and other performance metrics.

Also it is important to have a large number of nodes since this is more realistic to real vehicle system which would load the channel considerably.

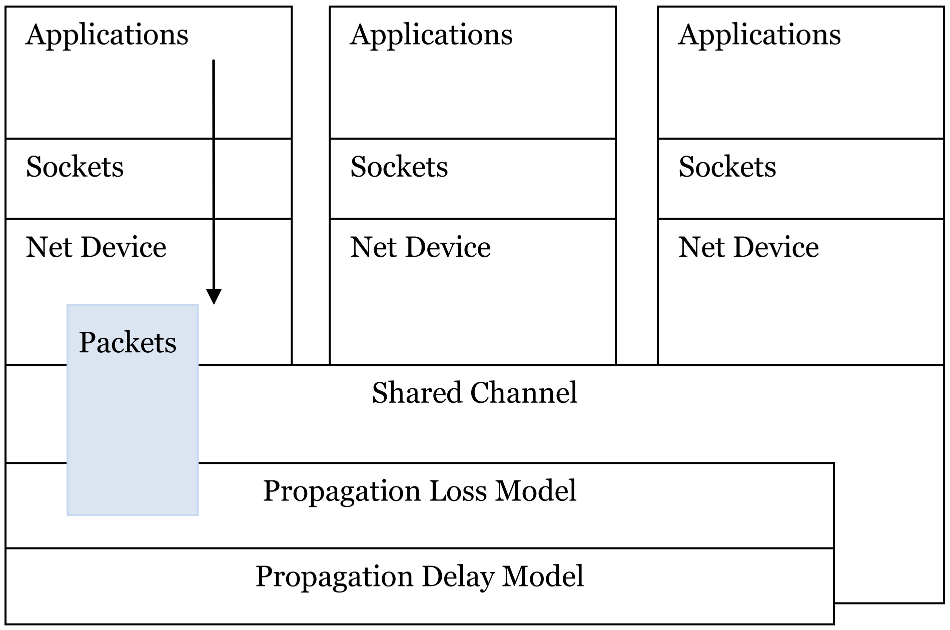
Both using PhySIM and a large number of nodes is computationally expensive. This project only attempts to make some benchmark simulations that can be used for comparison and also to visualize the current implementation of the Lower Layer communications of the Wifi Model in Ns3.

Issues with NS3 simulations

There are a few issues with ns3 discrete event simulations as described in reference [9].

1. The current ns-3 implementation of the Wifi channel has issues with channel capture, an important characteristic of real wireless communication.

The following is a simplification of a ns3 system but shows all its main parts:



When a frame is detected by the a net device, in this case a YansWifiPhy the program treats other packets that are late as noise even if they have more signal power. This is sort of offset by the fact that the packet reception is subdivided into chucks which are evaluated by themselves. For this reason channel capture phenomenon is not implemented in ns3 but is an important consideration when simulating a VANET scenario.

2. Nodes have no physical properties. A vehicle scenario would have to include the size and shape of vehicles including the type of antenna used. An omni-directional antenna is probably the most applicable but there still exists the issue with car shape since big trucks and busses can cause signal shadowing.

Experiment Setup

For this project I utilized the statistical framework in the ns-3 package. It was developed to support simulations that varied parameters and required a standard means of saving data about simulation runs and accessing them later for analysis and visualization.

The figure below shows this setup and the individual parts:

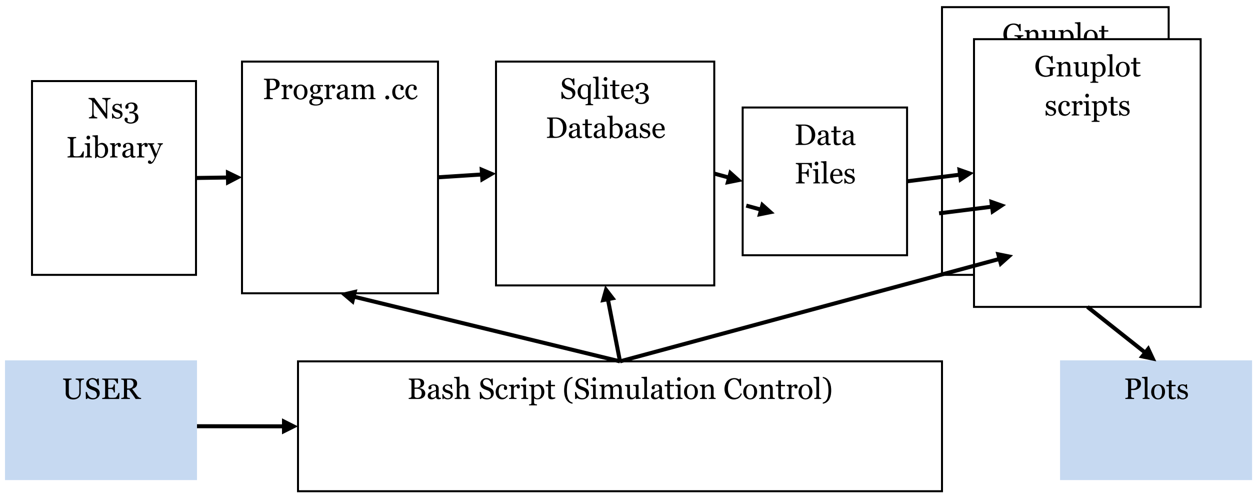


Figure: *The simulation setup for this project utilizing the statistical framework developed by ns3 contributors*

I worked from a template provided with the ns3.13 package but had to modify it to work for my purposes.

The wifi-example-sim program collects simulation data by using the DataCollector which at the end of a simulation outputs data to the sqlite3 database. The contains of the database can be viewed by the following:

```
$ sqlite3 data.db

sqlite> .dump
```

A part of the output is shown:

```
...

INSERT INTO "Singletons" VALUES('run-50-1','node[0]','wifi-tx-frames',31);

INSERT INTO "Singletons" VALUES('run-50-1','node[1]','wifi-rx-frames',31);

INSERT INTO "Singletons" VALUES('run-50-1','node[0]','sender-tx-packets',30);

INSERT INTO "Singletons" VALUES('run-50-1','node[1]','receiver-rx-packets',30);

INSERT INTO "Singletons" VALUES('run-50-1','node[0]','tx-pkt-size-count',30);

INSERT INTO "Singletons" VALUES('run-50-1','node[0]','tx-pkt-size-total',1920);

INSERT INTO "Singletons" VALUES('run-50-1','node[0]','tx-pkt-size-max',64);

INSERT INTO "Singletons" VALUES('run-50-1','node[0]','tx-pkt-size-min',64);

INSERT INTO "Singletons" VALUES('run-50-1','node[0]','tx-pkt-size-sqrsum',122880);

INSERT INTO "Singletons" VALUES('run-50-1','node[0]','tx-pkt-size-stddev',0);

INSERT INTO "Singletons" VALUES('run-50-1','.','delay-count',30);
INSERT INTO "Singletons" VALUES('run-50-1','.','delay-total',4878312);

INSERT INTO "Singletons" VALUES('run-50-1','.','delay-average',162610);

INSERT INTO "Singletons" VALUES('run-50-1','.','delay-max',629498);

INSERT INTO "Singletons" VALUES('run-50-1','.','delay-min',108166);
```

The database has the following tables with the shown schema:

```
sqlite> .tables

Experiments Metadata Singletons

sqlite> .schema

CREATE TABLE Experiments (run, experiment, strategy, input, description text);

CREATE TABLE Metadata ( run text, key text, value);

CREATE TABLE Singletons ( run text, name text, variable text, value );
```

GNUPLOT file

Scripting gnuplot commands is really easy and help is available on the internet. I just want to note a task which is not very common in examples available:

The (x,y) pairs in the file are the distance [m] and time [ns]. Since it's more relevant to show time in micro seconds the following line outputs the datapoints correctly.

```
                plot "wifi-delay.data" using ($1):($2/1000) with lines title "Delay Plot"
```

4. NS-3 Simulations

Channel Model

In ns-3 the wifiChannel is characterized by its PropagationLossModel and its PropagationDelayModel.

The channel Loss model is the algorithm that calculates the Receiver Signal Power (usually in dBm) given a certain transmission distance.

Some algorithms are deterministic like the logarithmic attenuation, while others are random including the Nakagami-m model.

The YansWifi model provides connection to the Nakagami propagation loss model which is meant to simulate fast-fading. The Nakagami model in ns3 is called:

ns3::NakagamiPropagationLossModel

There are a number of other models available also.

Contained in the ***propagation*** module is an example program called main-propagation-loss.cc and can be called from waf by the program-call main-propagation-loss. I modified this program to generate propagation losses for 25 dBm transmitter power and to then to output the gnuplot dataset to a file instead of std::cout. The gnuplot API is contained in the ***tools*** module.

The 3D plots generated aren't very clear so I opted to present one typical run that shows the typical fast-fading channel characteristic that should be expected.

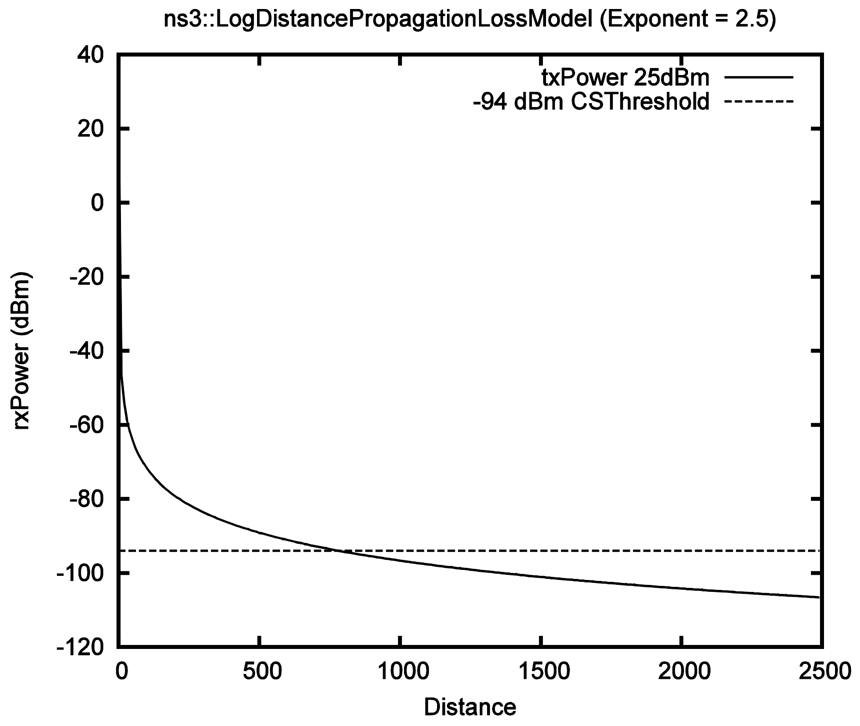


Figure: *The Log Channel Model is unrealistic in fast-fading systems. The Threshold is reached around 800m-1000m which is also the specified range of IEEE 802.11p communication*

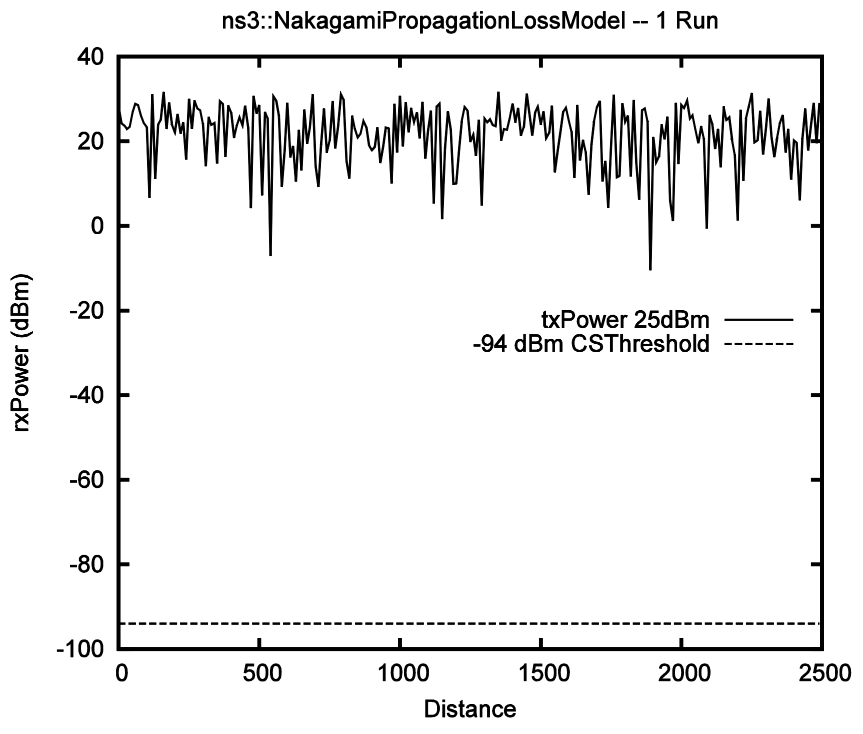


Figure: *Typical ns3::Nakagami-m plot with transmitter set at 25dBm*

The Nakagami-m model by itself does not decrease over distance so it has to be linked with a log function. For example it can be linked to the Three-Log Distance Model. The Three-Log Distance Model is defined by four regions: first region has no attenuation and three regions have exponential power attenuation. Default parameters are exponent of 3.8 for the middle and far regions and 1.9 for the near region. By chaining the three-log model with the Nakagami-m model the final result is a semi-realistic fast fading channel model.

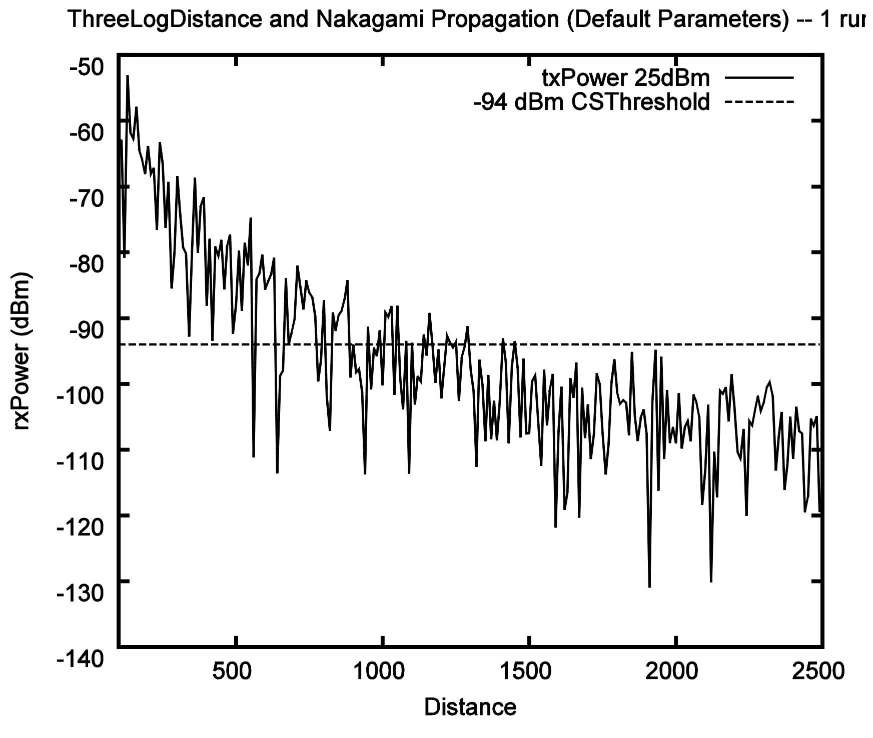


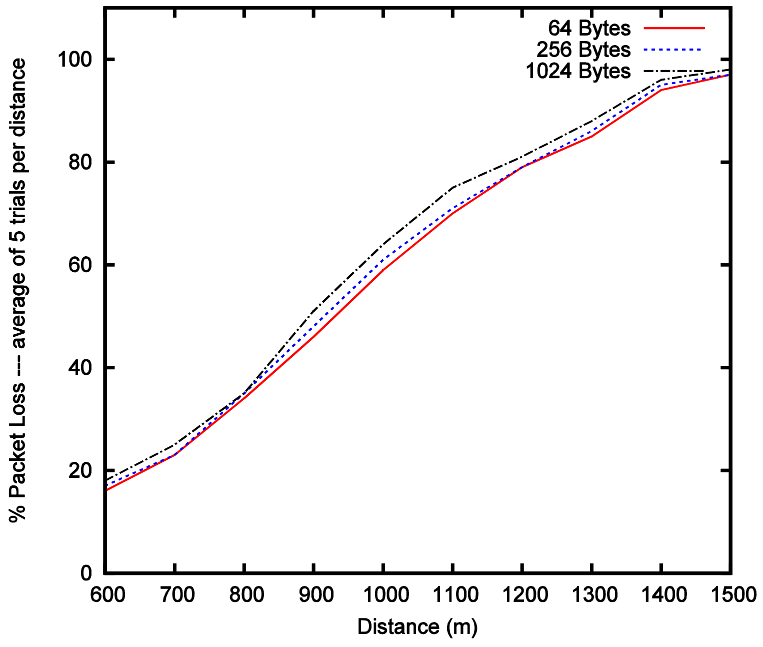
Figure: *The fast-fading channel model used in this project*

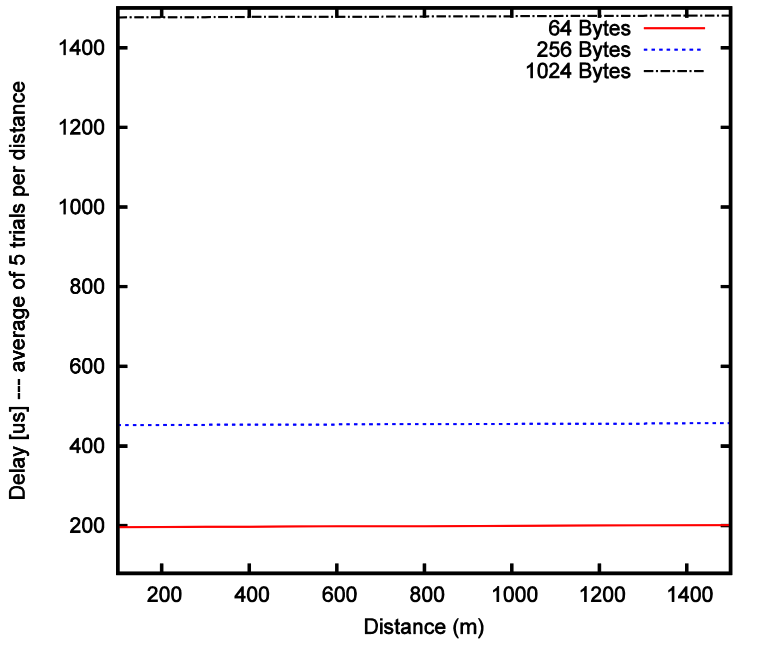
The Propagation Delay model was set to the constant speed model.

ns3::*ConstantSpeedPropagationDelayModel*

Scenario 1 – Two Node Simulation

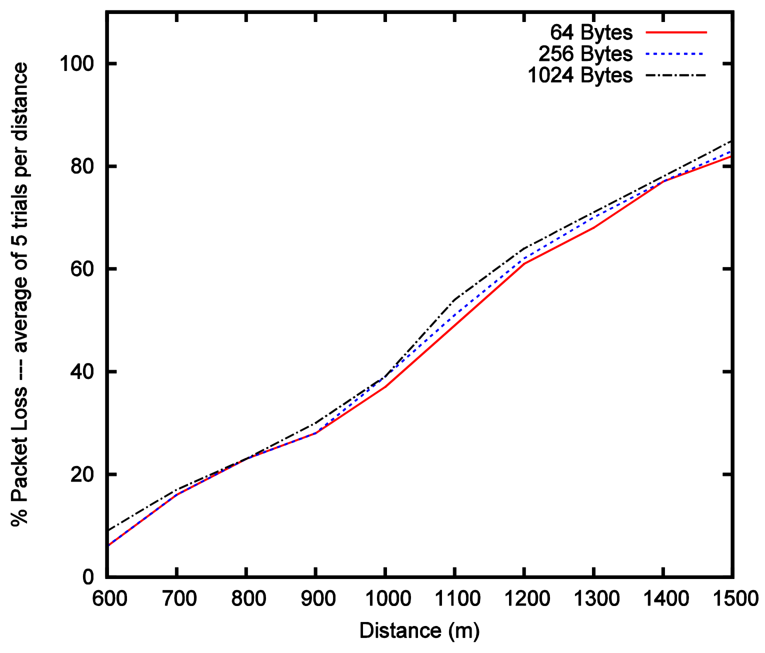
The first setup involved two nodes a fixed distance apart. One node was made a Beacon source running a constant Beacon transition application broadcasting to "255.255.255.255" every 0.5 seconds and the other a data sink. The data sink wifiPhy Net Device receiver was set to -94 dBm receive sensitivity (CSThreshold = EnergyThreshold). Both 802.11a and 802.11p were compared.





Figures: *Packet Loss Percentage Plot and Delay using wifi standard 802.11**a***

The IEEE 802.11p WIFI parameters produced a similar plot but had better packet-loss performance.

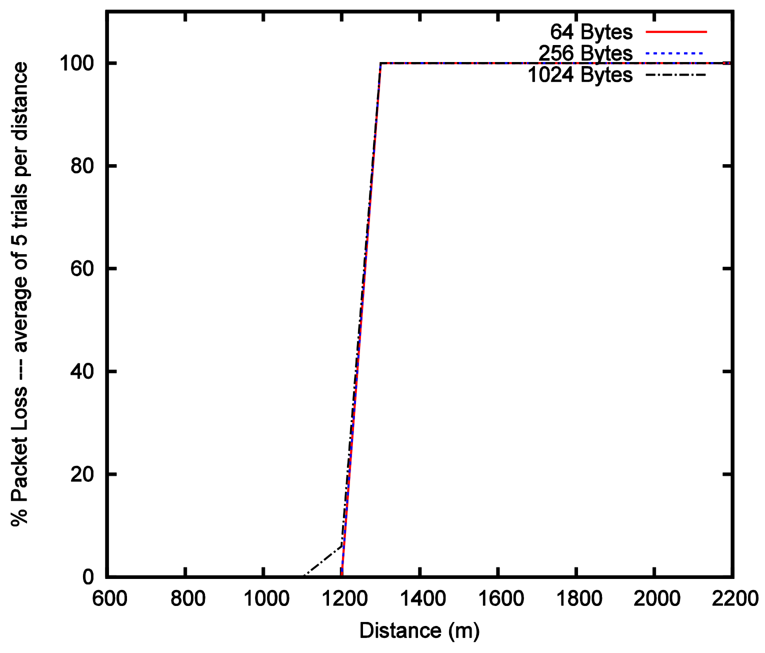


Figures: *Packet Loss Percentage Plot using wifi standard 802.11**p***

The delay plot was the same as is expected since the delay calculation is using the constant delay model. Another reason is because there is no medium contention since the sender is the only node utilizing the channel.

Message size of the beacon had a small effect on Packet-loss but there was a reasonable difference in the delay plot. This is of course expected since a larger message takes more time to be transmitted. In most of the reference papers the packet size (also equal to the frame size) is set at 250 Bytes.

Also just for reference a plot using just the log function is shown:



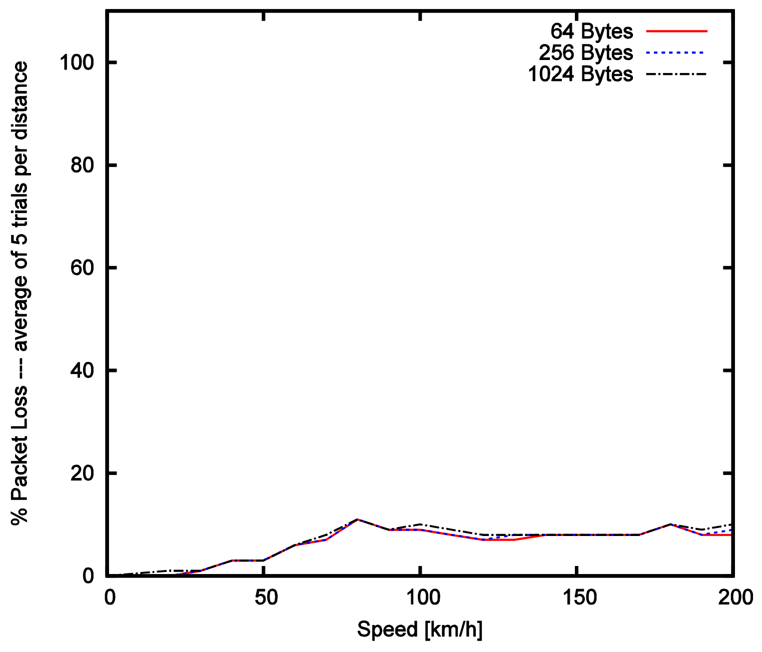
*Figures: Packet Loss Percentage Plot using wifi standard 802.11**p** and using log channel model*

As this plot shows this is probably an unrealistic result since fast-fading should cause some packet-loss well before the logarithmic boundary around 1000m.

Scenario 2 – Two Node Simulation in Random Walk in a 1km by 1km area

Scenario 2 was a setup to see the effect of speed on Packet Loss and Inter-beacon delay. A random-walk model is setup with a bound area of 1000m by 1000m. This area should allow the two nodes to be within communication range almost always.

Speed was varied from 0-200 km/h and direction was randomly changed every 10 seconds. The simulation was run for 200 seconds. The following plot shows the results.



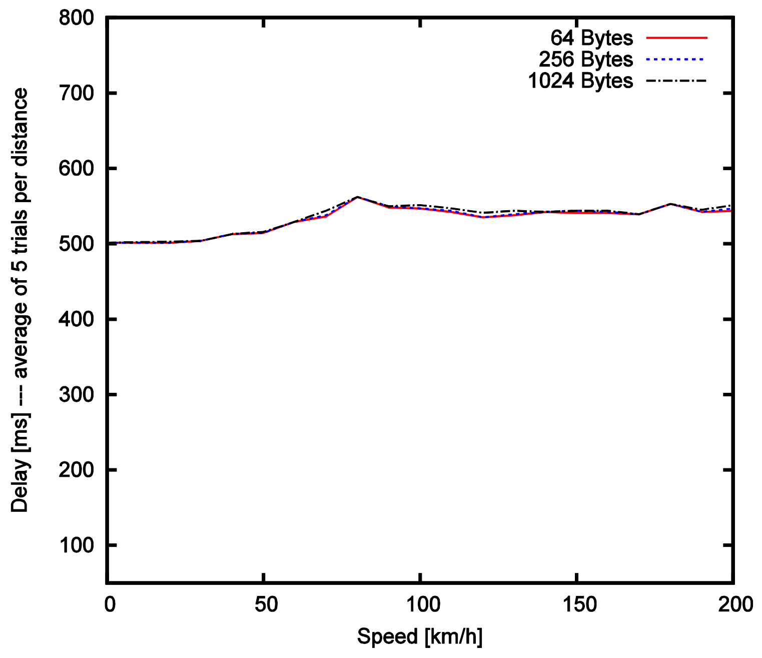


Figure: Packet Loss and Inter Beacon Delay as effected by mobile speed.

Speed seems to affect the packet loss in this random walk setup. Since the packets are being sent at 0.5 second intervals and the area is bound there is most likely a resonant speed or a set of resonant speeds that increase the packet loss.

As for inter-beacon delay there was also an effect due to node speed. The implementation for calculating inter-beacon delay is to record the time of a successful beacon reception and compare that to the last reception time.

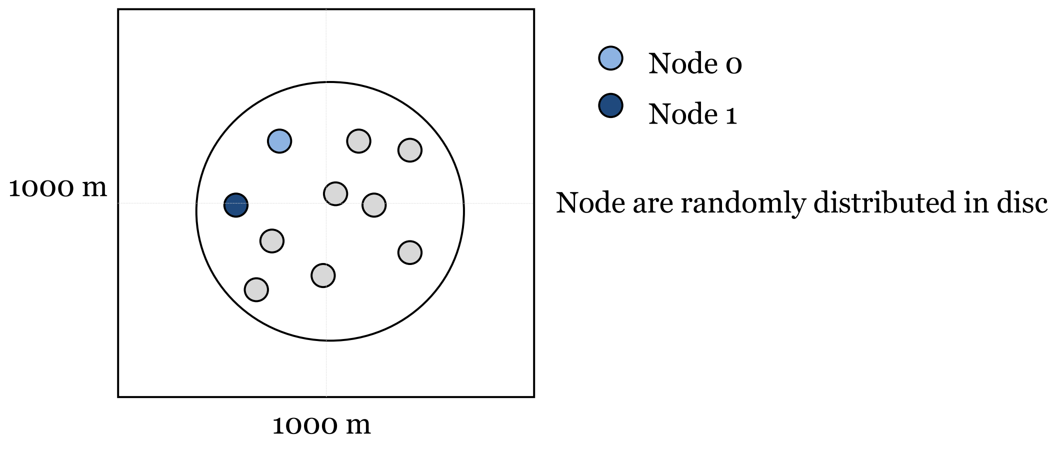
Also notice there is no significant different due to packet size so from now on the packet size is fixed to 256 Bytes.

Scenario 3 – Node Density in Random Walk

Next more nodes were added to the simulation to test the effect of node density. Node number is varied from 2 to 100 nodes. This simulation time was set to 200 seconds just like before.

The increased node number increased the complexity of the simulation which increased the simulation time from approximately 1 min to 40 mins for four different speeds.

The figure below shows graphically the starting random distribution of the nodes within a disc with radius 500 m.



The simulation tracked the packet reception on Node 1 due to Node 0. Packet Loss and Inter-Beacon delay is calculated as before.

A bug encountered in ns-3 is that application sender sources could not start at the same exact time. For this reason, a uniform random variable generator was used to randomly offset the sender application starting time.

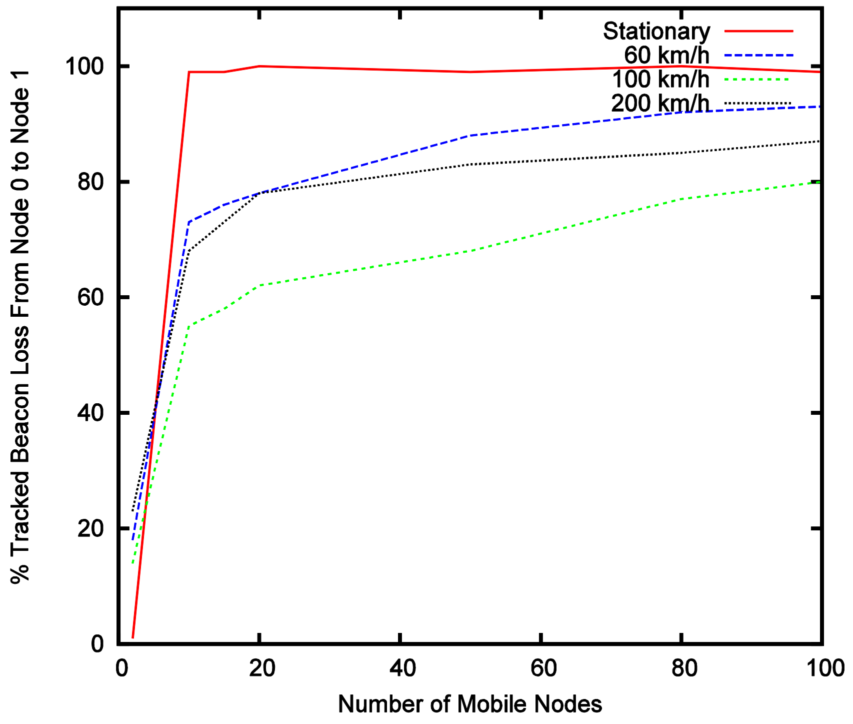


Figure: *Node Density effect and plots of different node speeds.*

The figure shows that at 100 km/h the Beacon loss is minimized. This again can be the result of a resonant speed for the given area and beacon interval.

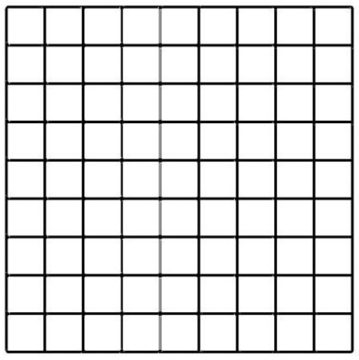
The effect of node density is that as more nodes are added the channel becomes saturated due to contention.

Scenario 4 – Using a Semi-Realistic Trace for Mobility within the 1km by 1km area

Originally mobile nodes where then made to move randomly in a 1 km x 1 km rectangle. This is unrealistic in that vehicles move on roads and in semi-predicable directions.

The next section describes how to generate Mobility Traces but here a more simple Trace is generated also with the SUMO traffic generator.

A grid pattern of 100m blocks in generated using the **netgen** tool provided with the SUMO package.



Each junction has an intersection and the street has two lanes and speed is set to approximately 60 km/h.

Ns2 traces files are generated from the state file and net file using the TraceExporter.jar program provided with the SUMO package.

In Ns3 the traces can be linked to the Mobility models for the given nodes in the global NodeList with the use of:

ns3::Ns2MobilityHelper

The following plot shows the results at 60 km/h.

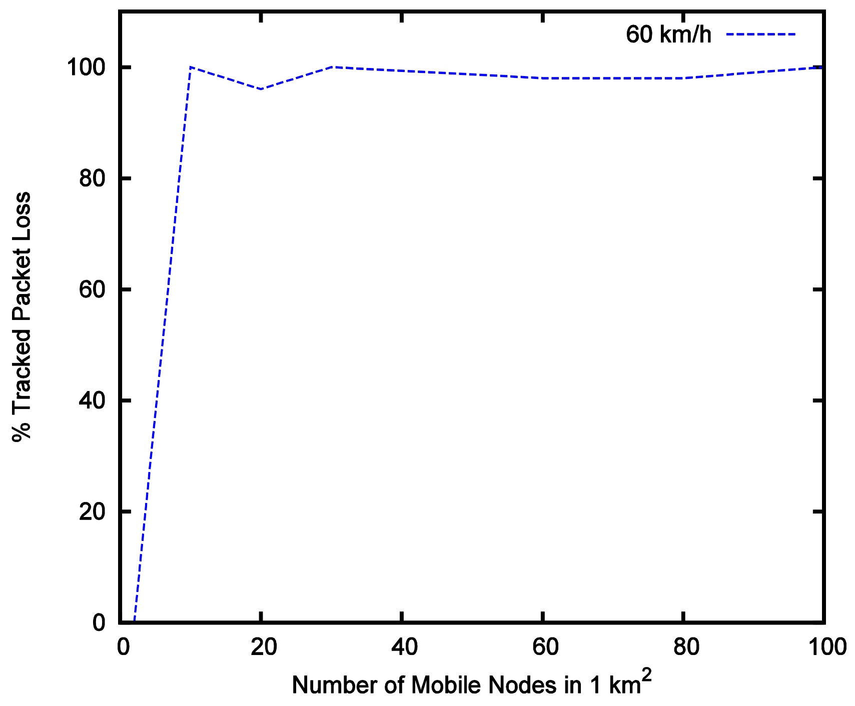


Figure: *Plot shows the result of using mobility traces from a grid street pattern*

To confirm that node 1 is actually moving, its course changes are outputted to a log file. These are triggered by Node 1 Mobility object and not the input traces themselves.

The plot above shows that more realistic movement puts more constraint on the communication between Node 0 and Node 1. Since the mobile nodes cannot move in a full random set of directions and are constrained to the grid pattern streets there is more contention as the nodes are closer to each other.

5. Generating Mobility Traces

This section describes how to generate traces to use in Ns3. This project did not utilize such a large trace since simulation time would have be too large.

In current research the use of realistic vehicular mobility is common. Some research has even used dynamic mobility in which the actual communication between vehicles affects their movement.

The first step is to produce a transportation infrastructure map. Using OpenStreetMap, a free open-source database of geographical information system data, a geographical realistic section of Lower Mainland roadway was exported. The following figure shows the section chosen for example:

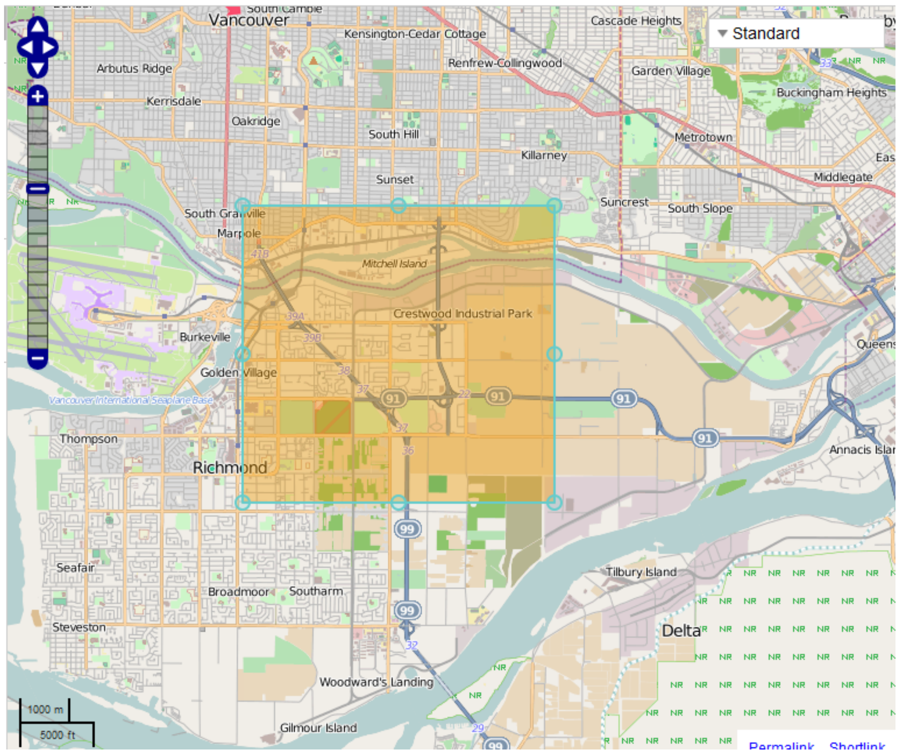


Figure: *Screen capture of OpenStreetView showing the area used for generating the infrastructure network*

OSM file format

OpenStreetMap exports data into xml files that are formatted based specification explained at the following webpage:

http://wiki.openstreetmap.org/wiki/OSM_XML

All nodes are specified by their geographical longitude and latitude. Sets of points can define polygon areas or roads/ways with lanes. As well meta-data is embedded within the file which explains the road rules and other characteristics of the objects on the map.

Importing into SUMO

Once data is collected as an OSM file it is imported into SUMO. The SUMO package has functionality for importing OSM files.

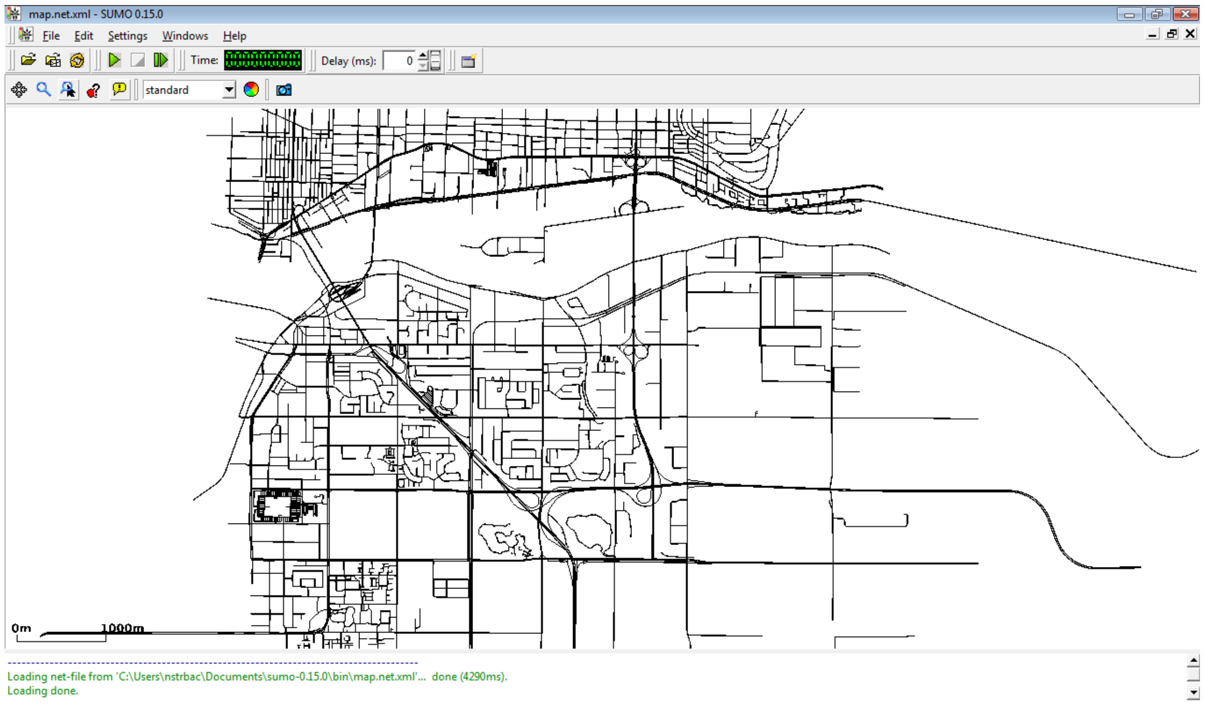
More information about importing OpenStreetMap files into SUMO is available at

http://sumo.sourceforge.net/doc/current/docs/userdoc/Networks/Import/OpenStreetMap.html

Shell command is:

```
netconvert --o file_input.osm -o file_output.net.xml
```

Using the SUMO GUI the traffic infrastructure can be viewed.



```
--------------------------------------------------------------------------------
Loading net-file from 'C:\Users\nstrbac\Documents\sumo-0.15.0\bin\map.net.xml'...  done (4290ms).
Loading done.
```

Figure: *Screen-capture showing the mobility network the SUMO GUI*

This is a map of approximately 8 km by 8 km and contains a broad spectrum of road density as can be seen.

Vehicular infrastructure is not simply just a set of geographical lines. Each roadway has certain properties like number of lanes, speed-limits and traffic rules.

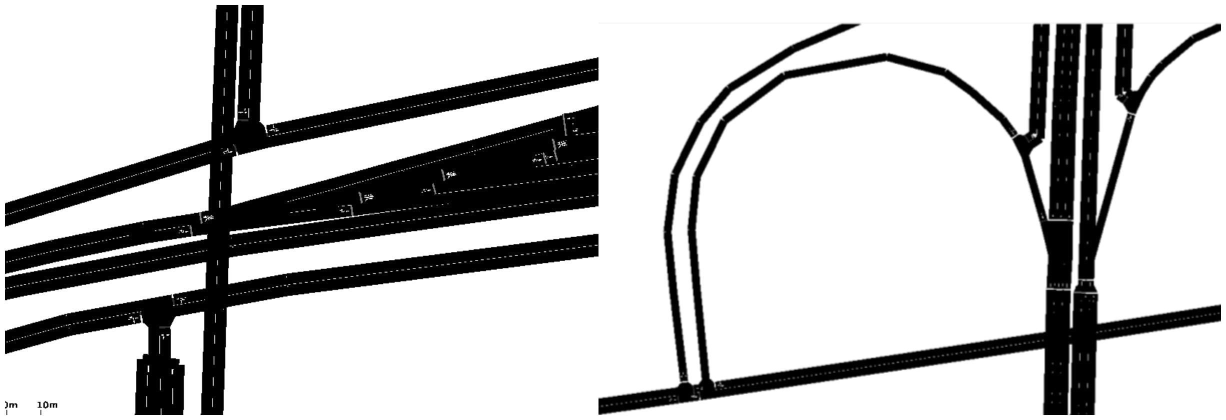


Figure: *Zoomed-in views of the transport infrastructure showing details*

Generating the Demand file

A traffic demand file needs to be combined with the network file to produce the traffic simulation. Current random routing algorithms are considered unrealistic but for the scope of this project it will suffice.

SUMO comes with a set of tools developed by the SUMO community. One of these is called *randomTrips.py* and is contained under <SUMO_HOME>\tools\trips.

Trips are just sources and destinations and are inputs to the router which calculates the actual route of the vehicles.

In <SUMO_HOME>\tools\trip the following produced the random trips:

```
randomTrips.py -n input_net.net.xml -e 1000 -l
```

This would produce 1000 trips for the inputted network. The trips are distributed evenly by trip length with the –l option.

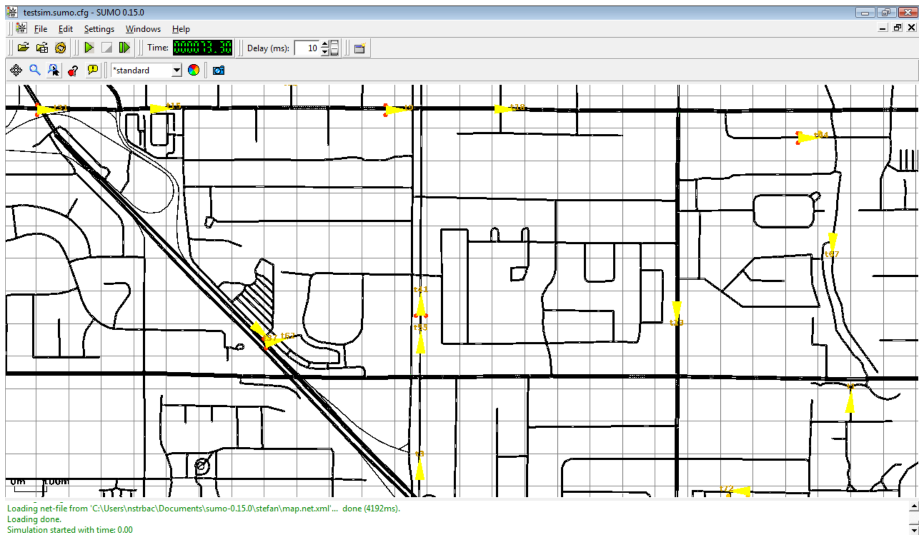
Next using durouter.exe:

```
Duarouter.exe -n input_net.net.xml -t trips.trips.xml -o routetest.rou.xml --ignore-errors --
repair
```

The --ignore-errors option is important since the random trips generated don't necessarily have to exist when roads are not connected.

Finally a configuration file is created to load the routes and network map into a simulation.

The result can be viewed in the SUMO GUI.



```
Loading net-file from 'C:\Users\nstrbac\Documents\sumo-0.15.0\stefan\map.net.xml'...  done (4192ms).
Loading done.
Simulation started with time: 0.00
```

Figure: *Here the size of the cars is exaggerated by 15 times for ease of viewing.*

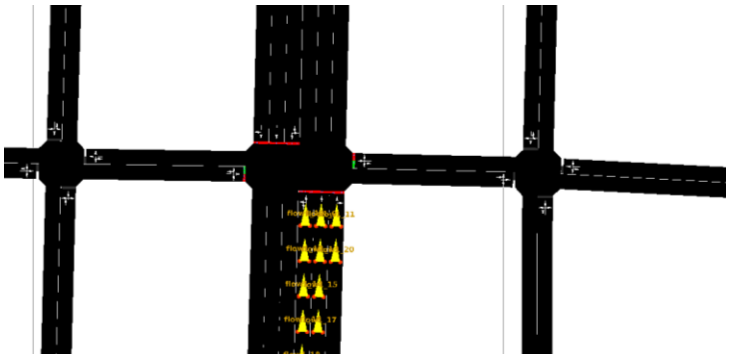


Figure: *Shows cars are waiting at an intersection*

The final step is to produce a simulation state file from which ns2 traces would be made. To do this, you would use TraceExporter.jar program. For example you would run TraceExporter from the cmd shell by:

```
java jar TraceExporter.jar ns2 …<options and input files specified here>
```

The output would produce ns2 traces would can be used in ns3 by using the ns3:Ns3MobilityHelper class. The current use of this object is to assign the traces to the global NodeList which holds all the nodes indexed from 0 to numberOfNodes. For more simulations this is enough.

6. Future Work

Most of the research in the area of V2V networks has not utilized ns3 mainly because it is still new and in development. The use of the parameters and Wifi implementation of IEEE 802.11p in ns3 is unconfirmed.

There are three main expansions which would be a reasonable project for ensc 427:

1. This project can be expanded to include a more realistic trace as described in the last section. Again given the limited information about the inter-road areas and buildings the communication channel cannot simulate reality deterministically.

2. Integrate the more detailed PhySim Channel model into this simulation. As pointed out in reference [9], the drawback of a deterministic model is increased simulation effort. The PhySim channel increases the computation time significantly.

3. Integrate the QoS functionality already provided in the WifiMAC implementation into a simulation scenario. Reference [10] had performed a QoS simulation using ns2 but its results can be used as a comparison and a starting point for programming.

7. References

Standards

[1] IEEE Std. 802.11p-2010, *Part 11: Wireless LAN Medium Access Control and Physical Layer Specifications*, IEEE, 2011

[2] IEEE 1609.1/2/3/4, *Trial-Use Standard for Wireless Access in Vehicular Environments (WAVE)*, IEEE Intelligent Transportation Systems Council, 2006

Presentations

[3] Dr. Thomas Strang, Matthias Röckl, "Vehicle Networks, V2X communication protocols", [Online] http://www.sti-innsbruck.at/fileadmin/documents/vn-ws0809/11-VN-WAVE.pdf [Accessed March 2012]

[4] Marc Werner, Radu Lupoaie,Sundar Subramanian, Jubin Jose, "MAC Layer Performance of ITS G5", [Online] http://www.etsi.org/WebSite/NewsandEvents/2012_ITSWORKSHOP.aspx [Accessed April 2012]

Papers

[5] Nicholas Loulloudes **,**"Evaluation of Vehicular Ad Hoc Network protocol and applications through simulations", University of Cyprus, 2011

[6] Hervé Boeglen, Benoît Hilt, Pascal Lorenz, Jonathan Ledy, Anne-Marie Poussard, Rodolphe Vauzelle, "A survey of V2V channel modeling for VANET simulations", IEEE, 2011

[7] K. Bilstrup, E. Uhlemann, E. G. Strom, and U. Bilstrup, "Evaluation of the IEEE 802.11p MAC Method for Vehicleto-Vehicle Communication," in Vehicular Technology Conference, 2008. VTC 2008-Fall. IEEE 68th, 2008, pp. 1-5.

[8] T. Murray, M. Cojocari, and H. Fu, "Measuring the performance of IEEE 802.11p using ns-2 simulator for vehicular networks," in Proc. IEEE Int'l Conf. on Electro/Information Technology (EIT), Ames, IA, May 2008.

[9] Michelle Segata and Renato Lo Cigno, "Simulation of 802.11 PHY/MAC: A Quest for Accuracy and Efficiency", 2012 [Online] http://vnt.disi.unitn.it/publications/2012/wons-12-selo.pdf

[10] Ali J. Ghandour Marco Di Felice Hassan Artail, "Modeling and Simulation of WAVE 1609.4-based Multi-channel Vehicular Ad-Hoc Networks", WAVE 1609 Working Group, 2012 [Online] http://vii.path.berkeley.edu/1609_wave/

Thesis Papers

[10] Zheng Chen, "Tracking Vehicular Motion-Position Using V2V Communication", University of Waterloo, 2010 [Online] http://hdl.handle.net/10012/5447 [Accessed March 2012]

8. Code Sample

All the ns3 C++ programs, gnuplot scripts and mobility traces are available at www.sfu.ca/~ssa21/ENSC427. The follow is the main program used for Scenario 3 and 4:

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
/*
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation;
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 *
 *
 *--------------------------------------------------------------------------------------
------
 *
 * The following code is based on examples provided with the ns3.13 package.
 *
 *
 */

#include <ctime>
#include <iostream>
#include <fstream>
#include <sstream>

#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/mobility-module.h"
```

```
#include "ns3/ns2-mobility-helper.h"
#include "ns3/wifi-module.h"
#include "ns3/propagation-module.h"
#include "ns3/internet-module.h"

#include "ns3/stats-module.h"

#include "wifi-example-apps.h"

using namespace ns3;
using namespace std;

NS_LOG_COMPONENT_DEFINE ("WiFi80211pExperiment");

Time CurTime;


void TxCallback (Ptr<CounterCalculator<uint32_t> > datac,
                 std::string path, Ptr<const Packet> packet) {

  NS_LOG_INFO ("Frame transmitted");

  NS_LOG_INFO ("Sent frame counted in " <<
               datac->GetKey ());
  datac->Update ();
  NS_LOG_INFO ("End of Callback OK");
  // end TxCallback
}

void RxCallback (Ptr<CounterCalculator<uint32_t> > datac,
                 std::string path, Ptr<const Packet> packet) {

  NS_LOG_INFO ("Frame recieved ");
  NS_LOG_INFO ("Sent frame counted in " <<
               datac->GetKey ());
  datac->Update ();
  // end RxCallback
}

//void DelayCallback ( Ptr<TimeMinMaxAvgTotalCalculator> delayc,
//                   std::string path, Ptr<const Packet> packet) {
//
//  CurTime.SetResolution (Time::US);
//  CurTime = Simulator::Now();
//
//  NS_LOG_INFO ("Transmission delay added");
//  delayc->Update (CurTime);
//  // end DelayCallback
//}


//
// Prints actual position and velocity when a course change event occurs in the
Mobility Model
//
static void
CourseChange (std::ostream *os, std::string foo, Ptr<const MobilityModel> mobility)
{
  Vector pos = mobility->GetPosition (); // Get position
  Vector vel = mobility->GetVelocity (); // Get velocity
  // Prints position and velocities
  *os << Simulator::Now () << " POS: x=" << pos.x << ", y=" << pos.y
      << ", z=" << pos.z << "; VEL:" << vel.x << ", y=" << vel.y
      << ", z=" << vel.z << std::endl;
}


//-----------------------------------------------------------------------
//-- main starts here
//-----------------------------------------------------------------------
int main (int argc, char *argv[]) {


//For Debugging just comment these

//LogComponentEnable("WiFi80211pExperiment", LOG_LEVEL_ALL);
//LogComponentEnable("WiFiDistanceApps", LOG_LEVEL_INFO);


//General Parameters of the experiment
  double distance = 50.0;
  double PacketSize = 64;
  double nodeSpeed = 0;
  int numOfNodes = 2;
  string format ("omnet");

  string experiment ("wifi-numOfNodes-test");
  string strategy ("wifi-default");
  string input;
  string runID;

//-----------------------------------------------------------------
//Config::Set ns3 Defaults for this experiment
//-----------------------------------------------------------------

  // disable fragmentation for frames below 2200 bytes
  Config::SetDefault ("ns3::WifiRemoteStationManager::FragmentationThreshold",
StringValue ("2200"));
  // turn off RTS/CTS for frames below 2200 bytes
  Config::SetDefault ("ns3::WifiRemoteStationManager::RtsCtsThreshold", StringValue
("2200"));

  //Mobility Defaults
  Config::SetDefault ("ns3::RandomWalk2dMobilityModel::Mode", StringValue ("Time"));
  Config::SetDefault ("ns3::RandomWalk2dMobilityModel::Time", StringValue ("10s"));
//a block is usually traversed in 10 sec
  Config::SetDefault ("ns3::RandomWalk2dMobilityModel::Speed", StringValue
("Constant:15.0"));  // 15 m/s which is approx. 50 km/h
  Config::SetDefault ("ns3::RandomWalk2dMobilityModel::Bounds", StringValue
("0|1000|0|1000")); //Defines a box of 1 km^2


//-----------------------------------------------------------------
// Set up command line parameters used to control the experiment.
//-----------------------------------------------------------------
  {
    stringstream sstr;
    sstr << "run-" << time (NULL);
    runID = sstr.str ();
  }

  CommandLine cmd;
  cmd.AddValue ("distance", "Distance apart to place nodes (in meters).",
                distance);
  cmd.AddValue ("nodeSpeed", "Speed scalar of Nodes",
                nodeSpeed);
  cmd.AddValue ("numOfNodes", "Number of Nodes to simulate",
                numOfNodes);
  cmd.AddValue ("PacketSize", "Packet size to send (in bytes).",
                 PacketSize);
  cmd.AddValue ("format", "Format to use for data output.",
                format);
  cmd.AddValue ("experiment", "Identifier for experiment.",
                experiment);
  cmd.AddValue ("strategy", "Identifier for strategy.",
                strategy);
  cmd.AddValue ("run", "Identifier for run.",
                runID);
  cmd.Parse (argc, argv);

  if (format != "omnet" && format != "db") {
      NS_LOG_ERROR ("Unknown output format '" << format << "'");
      return -1;
    }

  #ifndef STATS_HAS_SQLITE3
  if (format == "db") {
      NS_LOG_ERROR ("sqlite support not compiled in.");
      return -1;
    }
  #endif

  {
    stringstream sstr ("");
    sstr << double(numOfNodes);
    input = sstr.str ();
  }



  //------------------------------------------------------------------
  //-- Create nodes and network stacks
  //------------------------------------------------------------------
  NS_LOG_INFO ("Creating nodes.");
  NodeContainer nodes;
  nodes.Create (numOfNodes);

  NS_LOG_INFO ("Installing WiFi and Internet stack.");

  WifiHelper wifi = WifiHelper::Default ();
  wifi.SetStandard (WIFI_PHY_STANDARD_80211p_CCH);             //WIFI Standard IEEE 
802.11p

  NqosWifiMacHelper wifiMac = NqosWifiMacHelper::Default ();
  wifiMac.SetType ("ns3::AdhocWifiMac");                       //Specify AdHoc MAC


  //-----------------------------------------------------------------
  //-- Create Channel and wifiPhy Net Devices
  //--------------------------------------------------------------

  YansWifiPhyHelper wifiPhy = YansWifiPhyHelper::Default ();
  wifiPhy.Set("TxPowerEnd", DoubleValue(25));
  wifiPhy.Set("TxPowerStart", DoubleValue(25));
  wifiPhy.Set("EnergyDetectionThreshold", DoubleValue(-94));
  wifiPhy.Set("CcaMode1Threshold", DoubleValue(-94));

  Ptr<YansWifiChannel> wifiChannel = CreateObject <YansWifiChannel> ();

  //Create PropagationLossModels and DelayModel
  Ptr<ThreeLogDistancePropagationLossModel> log3 =
CreateObject<ThreeLogDistancePropagationLossModel> ();
  Ptr<NakagamiPropagationLossModel> nak = CreateObject<NakagamiPropagationLossModel>
();
  log3->SetNext (nak);

  wifiChannel->SetPropagationLossModel (log3);
  wifiChannel->SetPropagationDelayModel (CreateObject
<ConstantSpeedPropagationDelayModel> ());

  wifiPhy.SetChannel(wifiChannel);

  NetDeviceContainer nodeDevices = wifi.Install (wifiPhy, wifiMac, nodes);  //This is
the created NetDevice Container


  //----------------------------------------------------------------
  //-- Assign IP addresses
  //----------------------------------------------------------------
  InternetStackHelper internet;
  internet.Install (nodes);
  Ipv4AddressHelper ipAddrs;
  ipAddrs.SetBase ("192.168.0.0", "255.255.255.0");
  ipAddrs.Assign (nodeDevices);


  //---------------------------------------------------------------
  //-- Setup Mobility for the nodes
  //---------------------------------------------------------------
  //NS_LOG_INFO ("Installing static mobility; distance " << distance << " .");
  //
  MobilityHelper mobility;
  mobility.SetPositionAllocator ("ns3::RandomDiscPositionAllocator",      //Inital
Positions are randomly around 500,500 a circle of 200m
                                 "X", StringValue ("500.0"),
                                 "Y", StringValue ("500.0"),
                                 "Rho", StringValue ("Uniform:0:200"));
  //
  //nodeSpeed = nodeSpeed*1000/3600;                                          //Speed
km/h convert to m/s and convert to string type
  //std::ostringstream oss;
  //oss << "Constant:"<< nodeSpeed;
  //std::string speedValue = oss.str();
  //NS_LOG_INFO ("Installing velocity: speed: " << speedValue << " .");
//
  //mobility.SetMobilityModel ("ns3::RandomWalk2dMobilityModel",            //Speed
and direction changes every 10 sec
  //                             "Mode", StringValue ("Time"),              //
Movement is bound in a box 1km by 1km
```

```
  //                           "Time", StringValue ("10s"),
  //                           "Speed", StringValue (speedValue),
  //                           "Bounds", StringValue ("0|1000|0|1000"));
  mobility.InstallAll ();                                             //Mobility
Model installed on all notes
//
 //
  //Its a good idea to record the movement so it makes sense
  //Thou this will increase the simulation time


  // Enable logging from the ns2 helper
  LogComponentEnable ("Ns2MobilityHelper",LOG_LEVEL_DEBUG);

  //open log file for output
  std::ofstream os;
  os.open ("movementLog.txt");

   //Configure callback for logging the movement
   Config::Connect ("/NodeList/1/$ns3::MobilityModel/CourseChange",
                   MakeBoundCallback (&CourseChange, &os));             //Specified
for all nodes but can just record for a subset


    Ns2MobilityHelper ns2 = Ns2MobilityHelper ("mobility.txt");
   ns2.Install (); // configure movements for each node, while reading trace file

  //-----------------------------------------------------------------
  //-- Create traffic source and sink
  //-----------------------------------------------------------------

  //Arrays
  Ptr<Node> tempNode [numOfNodes];
  Ptr<Sender> sender [numOfNodes];
  Ptr<Receiver> receiver [numOfNodes];

  UniformVariable uVar (0,0.001);             //Uniform Random Variable is needed
because the nodes can't start to generate
                                              //traffic at the same exact time.

  //The following adds a Beacon Generator App and a Beacon Sink App on each Node
  // Sender is App 0 and Reciever is App 1 on each node
  for(int ni=0; ni < numOfNodes; ni++)
  {

        tempNode[ni] = NodeList::GetNode (ni);

        //Add Sender Application
        sender[ni]= CreateObject<Sender>();
        sender[ni]->SetStartTime (Seconds (uVar.GetValue()));
        tempNode[ni]->AddApplication (sender[ni]);

        //Add Receiver Application
        receiver[ni] = CreateObject<Receiver>();
        receiver[ni]->SetStartTime (Seconds (uVar.GetValue()));
        tempNode[ni]->AddApplication (receiver[ni]);
  }
  NS_LOG_INFO ("Finished installing Apps on each node");

    Config::Set("/NodeList/*/ApplicationList/0/$Sender/PacketSize",
                UintegerValue(PacketSize));                              //Packet
Size is set here
    Config::Set("/NodeList/*/ApplicationList/0/$Sender/Destination",
                Ipv4AddressValue("255.255.255.255"));                   //Sets the
Sender Applications to Broadcast Addresses




  //------------------------------------------------------------------
  //-- Setup stats and data collection
  //------------------------------------------------------------------

  // Create a DataCollector object to hold information about this run.
  DataCollector data;
  data.DescribeRun (experiment,
                    strategy,
                    input,
                    runID);

  // Add any information we wish to record about this run.
  data.AddMetadata ("author", "ensc427");


  // Create a counter to track how many frames are generated.  Updates
  // are triggered by the trace signal generated by the WiFi MAC model
  // object.  Here we connect the counter to the signal via the simple
  // TxCallback() glue function defined above.
  Ptr<CounterCalculator<uint32_t> > totalTx =
    CreateObject<CounterCalculator<uint32_t> >();
  totalTx->SetKey ("wifi-tx-Frames");
  totalTx->SetContext ("node[0]");
  Config::Connect ("/NodeList/0/DeviceList/*/$ns3::WifiNetDevice/Mac/MacTx",
                   MakeBoundCallback (&TxCallback, totalTx));
  data.AddDataCalculator (totalTx);

  // This is similar, but creates a counter to track how many frames
  // are received.  Instead of our own glue function, this uses a
  // method of an adapter class to connect a counter directly to the
  // trace signal generated by the WiFi MAC.
  Ptr<PacketCounterCalculator> totalRx =
    CreateObject<PacketCounterCalculator>();
  totalRx->SetKey ("wifi-rx-Frames");
  totalRx->SetContext ("node[1]");
  //Config::Connect ("/NodeList/1/DeviceList/*/$ns3::WifiNetDevice/Mac/MacRx",
  //                 MakeBoundCallback (&RxCallback, totalRx));
   Config::Connect ("/NodeList/1/DeviceList/*/$ns3::WifiNetDevice/Mac/MacRx",
                   MakeCallback (&PacketCounterCalculator::PacketUpdate,
                                 totalRx));
  data.AddDataCalculator (totalRx);




  // This counter tracks how many packets---as opposed to frames---are
  // generated.  This is connected directly to a trace signal provided
  // by our Sender class.
  Ptr<PacketCounterCalculator> appTx =
    CreateObject<PacketCounterCalculator>();
  appTx->SetKey ("sender-tx-packets");
  appTx->SetContext ("node[0]");
  Config::Connect ("/NodeList/0/ApplicationList/0/$Sender/Tx",
                   MakeCallback (&PacketCounterCalculator::PacketUpdate,
                                 appTx));
  data.AddDataCalculator (appTx);




  // Here a counter for received packets is directly manipulated by
  // one of the custom objects in our simulation, the Receiver
  // Application.  The Receiver object is given a pointer to the
  // counter and calls its Update() method whenever a packet arrives.
  Ptr<CounterCalculator<> > appRx =
    CreateObject<CounterCalculator<> >();
  appRx->SetKey ("receiver-rx-packets");
  appRx->SetContext ("node[1]");
  receiver[1]->SetCounter (appRx);
  data.AddDataCalculator (appRx);




  /**
   * Just to show this is here...
   Ptr<MinMaxAvgTotalCalculator<uint32_t> > test =
   CreateObject<MinMaxAvgTotalCalculator<uint32_t> >();
   test->SetKey("test-dc");
   data.AddDataCalculator(test);

   test->Update(4);
   test->Update(8);
   test->Update(24);
   test->Update(12);
  **/


  // Here we directly manipulate another DataCollector tracking min,
  // max, total, and average propagation delays.  Check out the Sender
  // and Receiver classes to see how packets are tagged with
  // timestamps to do this.
  //Ptr<TimeMinMaxAvgTotalCalculator> delayStat =
CreateObject<TimeMinMaxAvgTotalCalculator>();
  //delayStat->SetKey ("delay");
  //delayStat->SetContext (".");
  //receiver[1]->SetDelayTracker (delayStat);
  //data.AddDataCalculator (delayStat);
  //delayStat->Enable();

  //Config::Connect ("/NodeList/1/DeviceList/*/$ns3::WifiNetDevice/Mac/MacRx",
                 //   MakeBoundCallback (&DelayCallback, delayStat));


  // Here we directly manipulate another DataCollector tracking min,
  // max, total, and average inter-Beacon delays.  Check out the Sender
  // and Receiver classes to see how packets are tagged with
  // timestamps to do this.
  Ptr<TimeMinMaxAvgTotalCalculator> beaconStat =
CreateObject<TimeMinMaxAvgTotalCalculator>();
  beaconStat->SetKey ("interbeacontime");
  beaconStat->SetContext (".");
  receiver[1]->SetBeaconTracker (beaconStat);
  data.AddDataCalculator (beaconStat);
  beaconStat->Enable();

 //Set Tracking Address of Reciever 1 to the following Address
 Config::Set("/NodeList/1/ApplicationList/1/$Receiver/SourceToTrack",
               Ipv4AddressValue ("192.168.0.1"));                      //Tracks
Sender 0 on Node 0




  //------------------------------------------------------------------
  //-- Run the simulation
  //------------------------------------------------------------------
  NS_LOG_INFO ("Run Simulation.");
  Simulator::Stop (Seconds (60.0));
  Simulator::Run ();



  //------------------------------------------------------------------
  //-- Generate statistics output.
  //------------------------------------------------------------------

  // Pick an output writer based in the requested format.
  Ptr<DataOutputInterface> output = 0;
  if (format == "omnet") {
      NS_LOG_INFO ("Creating omnet formatted data output.");
      output = CreateObject<OmnetDataOutput>();
    } else if (format == "db") {
    #ifdef STATS_HAS_SQLITE3
      NS_LOG_INFO ("Creating sqlite formatted data output.");
      output = CreateObject<SqliteDataOutput>();
    #endif
    } else {
      NS_LOG_ERROR ("Unknown output format " << format);
    }

  // Finally, have that writer interrogate the DataCollector and save
  // the results.
  if (output != 0)
    output->Output (data);

  // Free any memory here at the end of this example.
  Simulator::Destroy ();

  os.close (); //Close Mobility Logger

  // end main
}
```

END of DOCUMENT