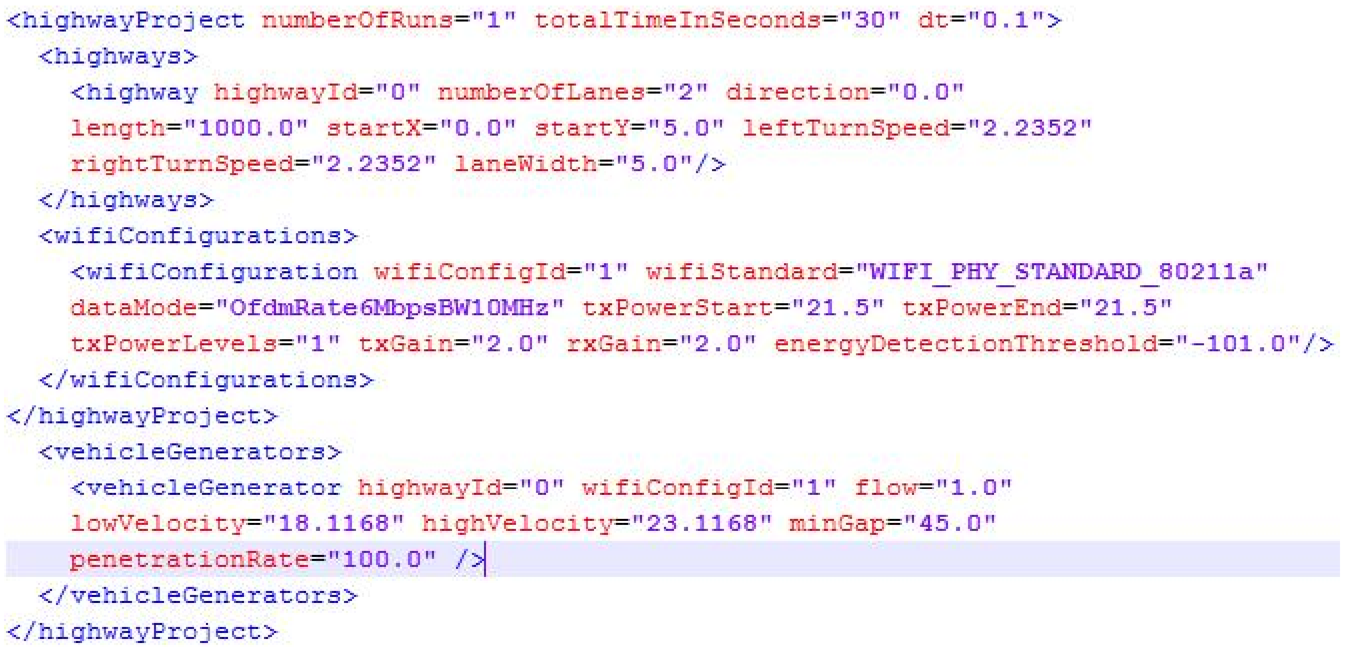
Simple Highway Example

The code described below is available as part of the source code download. A much more complicated example is included in the source code which creates a four-way intersection. That XML file includes connections and TrafficLightGenerators. The source-code for running it remains the same.

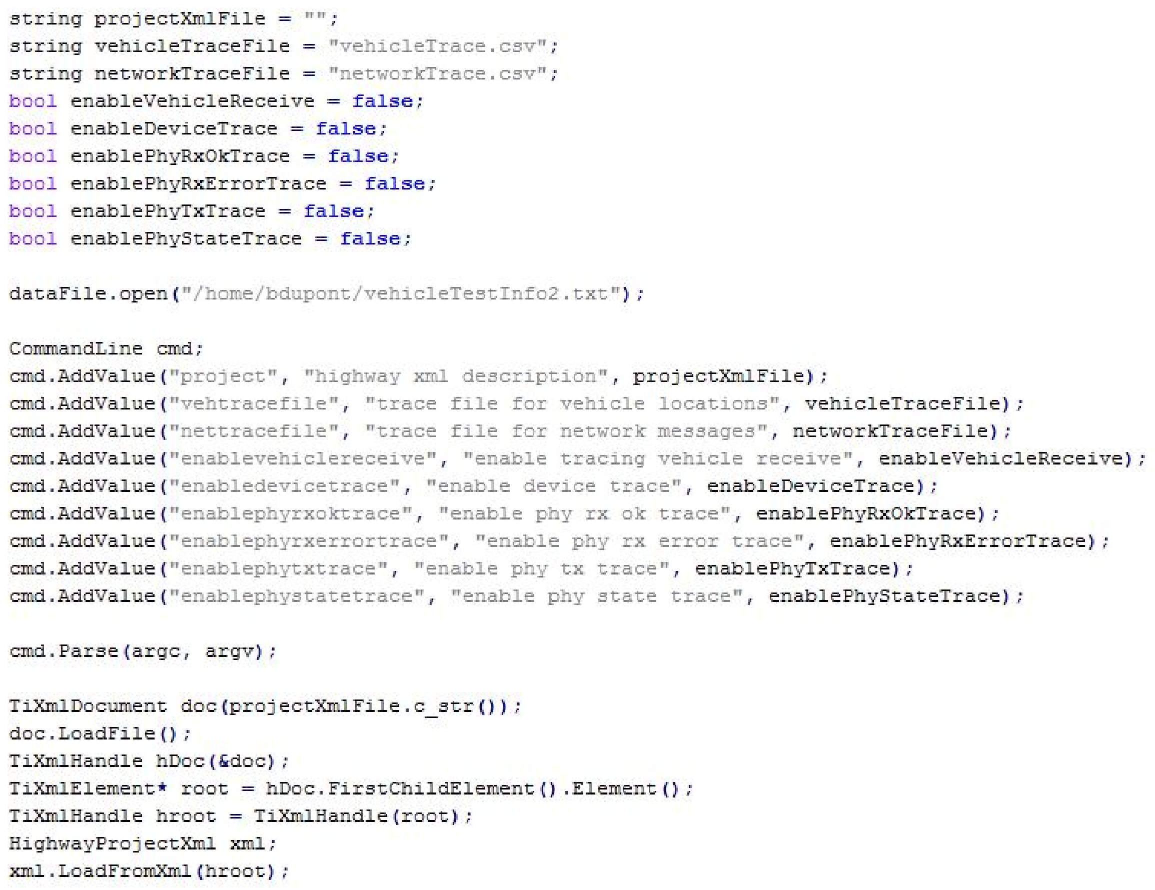
The first step is to create an XML document. Below is a simple example for a straight highway with Wifi configured and attached to a VehicleGenerator.



```
<highwayProject numberOfRuns="1" totalTimeInSeconds="30" dt="0.1">
  <highways>
    <highway highwayId="0" numberOfLanes="2" direction="0.0"
    length="1000.0" startX="0.0" startY="5.0" leftTurnSpeed="2.2352"
    rightTurnSpeed="2.2352" laneWidth="5.0"/>
  </highways>
  <wifiConfigurations>
    <wifiConfiguration wifiConfigId="1" wifiStandard="WIFI_PHY_STANDARD_80211a"
    dataMode="OfdmRate6MbpsBW10MHz" txPowerStart="21.5" txPowerEnd="21.5"
    txPowerLevels="1" txGain="2.0" rxGain="2.0" energyDetectionThreshold="-101.0"/>
  </wifiConfigurations>
</highwayProject>
  <vehicleGenerators>
    <vehicleGenerator highwayId="0" wifiConfigId="1" flow="1.0"
    lowVelocity="18.1168" highVelocity="23.1168" minGap="45.0"
    penetrationRate="100.0" />
  </vehicleGenerators>
</highwayProject>
```

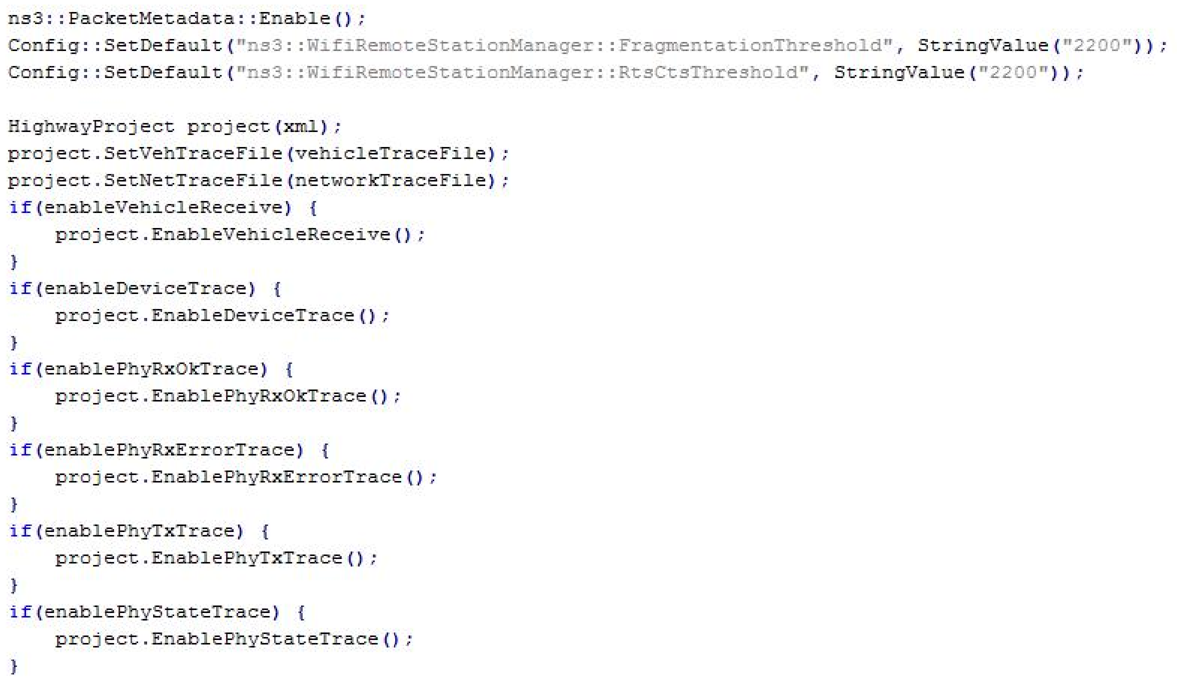
The root node contains how long the simulation will run and what the step size is. While "numberOfRuns" is defined in the XML, it is not currently used by HighwayProject. The "highways" root contains one or more "highway" nodes. For more details on what the attributes mean, see section **3.2.1**. The next section is "wifiConfigurations" and stores all of the wifi configuration instances. The “wifiConfigId” is only used to join “wifiConfiguration” instances to “vehicleGenerator” instances and is not available in the code. The last node is the “vehicleGenerators” node. It links to a Highway instance.

The second step is to create a main instance that will use this XML file.



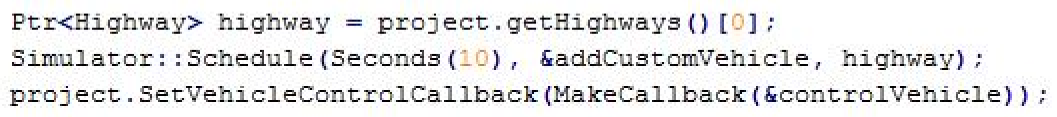
```
string projectXmlFile = "";
string vehicleTraceFile = "vehicleTrace.csv";
string networkTraceFile = "networkTrace.csv";
bool enableVehicleReceive = false;
bool enableDeviceTrace = false;
bool enablePhyRxOkTrace = false;
bool enablePhyRxErrorTrace = false;
bool enablePhyTxTrace = false;
bool enablePhyStateTrace = false;

dataFile.open("/home/bdupont/vehicleTestInfo2.txt");

CommandLine cmd;
cmd.AddValue("project", "highway xml description", projectXmlFile);
cmd.AddValue("vehtracefile", "trace file for vehicle locations", vehicleTraceFile);
cmd.AddValue("nettracefile", "trace file for network messages", networkTraceFile);
cmd.AddValue("enablevehiclereceive", "enable tracing vehicle receive", enableVehicleReceive);
cmd.AddValue("enabledevicetrace", "enable device trace", enableDeviceTrace);
cmd.AddValue("enablephyrxoktrace", "enable phy rx ok trace", enablePhyRxOkTrace);
cmd.AddValue("enablephyrxerrortrace", "enable phy rx error trace", enablePhyRxErrorTrace);
cmd.AddValue("enablephytxtrace", "enable phy tx trace", enablePhyTxTrace);
cmd.AddValue("enablephystatetrace", "enable phy state trace", enablePhyStateTrace);

cmd.Parse(argc, argv);

TiXmlDocument doc(projectXmlFile.c_str());
doc.LoadFile();
TiXmlHandle hDoc(&doc);
TiXmlElement* root = hDoc.FirstChildElement().Element();
TiXmlHandle hroot = TiXmlHandle(root);
HighwayProjectXml xml;
xml.LoadFromXml(hroot);
```

This section of the code is responsible for creating and parsing the command line parameters. Once they have been loaded, the XML document is loaded and passed into the utility bean HighwayProjectXml. That class and its children contain all configuration (and can be created by hand if desired).



```
ns3::PacketMetadata::Enable();
Config::SetDefault("ns3::WifiRemoteStationManager::FragmentationThreshold", StringValue("2200"));
Config::SetDefault("ns3::WifiRemoteStationManager::RtsCtsThreshold", StringValue("2200"));

HighwayProject project(xml);
project.SetVehTraceFile(vehicleTraceFile);
project.SetNetTraceFile(networkTraceFile);
if(enableVehicleReceive) {
    project.EnableVehicleReceive();
}
if(enableDeviceTrace) {
    project.EnableDeviceTrace();
}
if(enablePhyRxOkTrace) {
    project.EnablePhyRxOkTrace();
}
if(enablePhyRxErrorTrace) {
    project.EnablePhyRxErrorTrace();
}
if(enablePhyTxTrace) {
    project.EnablePhyTxTrace();
}
if(enablePhyStateTrace) {
    project.EnablePhyStateTrace();
}
```

This next section creates the HighwayProject instance and passes the XML configuration into it. It pulls the location for the vehicle and network trace and passes those in. Finally, depending on the boolean flags, the code enables various kinds of traces.

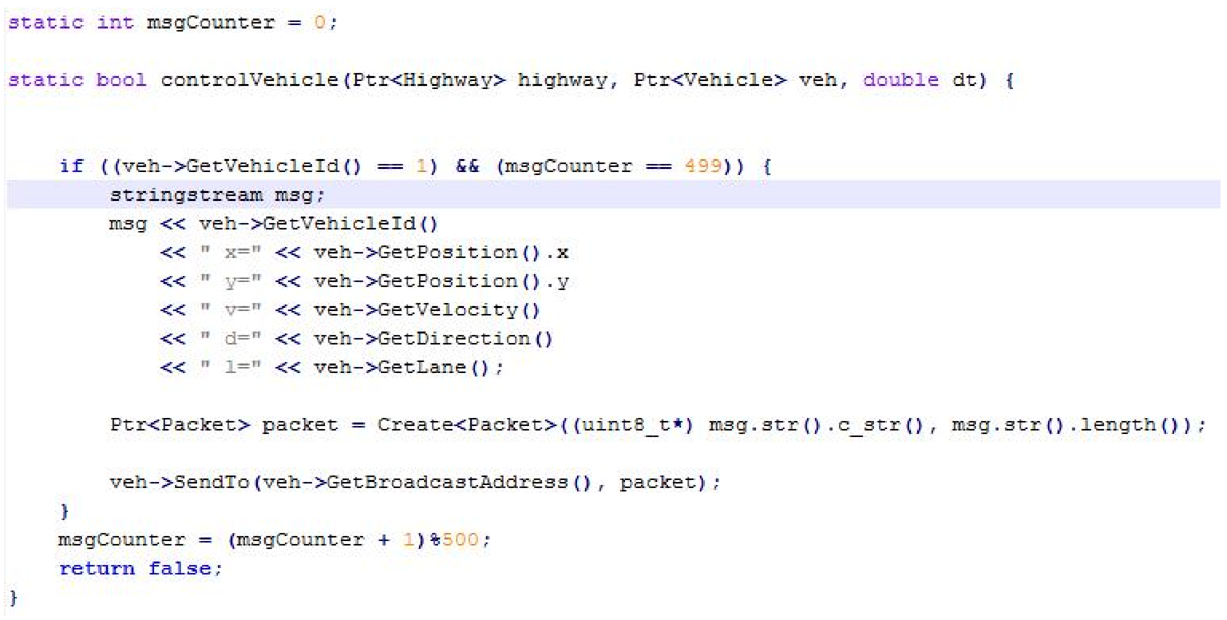


```
Ptr<Highway> highway = project.getHighways()[0];
Simulator::Schedule(Seconds(10), &addCustomVehicle, highway);
project.SetVehicleControlCallback(MakeCallback(&controlVehicle));
```

This section accomplishes some customization hooks. Note that “*project.getHighways()*” returns a map of highwayId (from the XML) to Ptr<Highway>. The Simulator::Schedule demonstrates how to schedule a function to be called by the simulator. The next line also shows how to establish a callback. The definition of “*addCustomVehicle*” and “*controlVehicle*” are displayed below.

```
static void addCustomVehicle(Ptr<Highway> highway) {
    int id = -1;
    int lane = 1;
    double speed = 40.0;

    Ptr<Model> model = CreateObject<Model>();
    model->SetDesiredVelocity(speed);
    model->SetDeltaV(4.0);
    model->SetAcceleration(0.5);
    model->SetDeceleration(3.0);
    model->SetMinimumGap(2.0);
    model->SetTimeHeadway(0.1);
    model->SetSqrtAccelerationDeceleration(sqrt(model->GetAcceleration() * model->GetDeceleration()));

    Ptr<LaneChange> laneChange = CreateObject<LaneChange>();
    laneChange->SetPolitenessFactor(0.2);
    laneChange->SetDbThreshold(0.3);
    laneChange->SetGapMin(2.0);
    laneChange->SetMaxSafeBreakingDeceleration(12.0);
    laneChange->SetBiasRight(0.2);

    Ptr<Vehicle> temp=CreateObject<Vehicle>();
    temp->SetVehicleId(id);
    temp->IsEquipped=false;
    temp->SetModel(model);
    temp->SetLaneChange(laneChange);
    temp->SetLength(4);
    temp->SetWidth(2);
    temp->SetVelocity(speed);
    temp->SetAcceleration(0.0);
    temp->SetLane(lane);
    temp->SetDirection(0.0);
    temp->SetPosition(highway->GetLaneStart(lane));
    highway->AddVehicleToBeginning(temp);
}
```

This section of code demonstrates how to create a Vehicle instance and adding to a Highway instance. Note that the line "*temp->SetPosition*" is not strictly required. The function "*AddVehicleToBeginning"* sets the Vehicle position already.



```
static int msgCounter = 0;

static bool controlVehicle(Ptr<Highway> highway, Ptr<Vehicle> veh, double dt) {


    if ((veh->GetVehicleId() == 1) && (msgCounter == 499)) {
        stringstream msg;
        msg << veh->GetVehicleId()
            << " x=" << veh->GetPosition().x
            << " y=" << veh->GetPosition().y
            << " v=" << veh->GetVelocity()
            << " d=" << veh->GetDirection()
            << " l=" << veh->GetLane();

        Ptr<Packet> packet = Create<Packet>((uint8_t*) msg.str().c_str(), msg.str().length());

        veh->SendTo(veh->GetBroadcastAddress(), packet);
    }
    msgCounter = (msgCounter + 1)%500;
    return false;
}
```

This is a control vehicle callback function. This is called on every single Vehicle for every single Highway configured. The counter is used to make sure that messages are not flooded. There seems to be an error if messages are sent from the Vehicle too soon. If the counter limit is set to 0 (*msgCounter* == *0*), ns-3 sends an error. The function always returns false because the Vehicle acceleration and position data is not modified.