SOFTWARE REQUIREMENT SPECIFICATION

The Software Requirement Specification is the description of the software system. It includes the functional and non-functional requirements to perform specific goals. It also provides the use cases for user interaction that software must provide. It is the end of requirement engineering process.

VANET - Vehicular ad Hoc Network enables exchange of information from vehicle-to-vehicle and vehicle-to-Road Side Unit (RSU). Information is conveyed to the destination through single or multiple hops. It uses a routing protocol to forward the packets with maximum lifetime and less delay.

It also describes project for implementing major application of VANET- Smartphone integrated driving safety application along with a traffic signal priority control method also to clear the path for emergency vehicle is modeled.

The project can be implemented with the following hardware and software requirements:

1.1 Functional Requirements:

The Functional Requirements defines the function of a system which describes set of inputs, the behavior, and its outputs to carry out the project smoothly. Below is the functional hardware requirements considered for carrying out simulation:

System	:	Personal Computer or Laptop
Processor	:	i3 Processor with 1.90GHz
RAM	:	4 GB
Secondary memory	:	500 GB

The other hardware requirements for the system development are as given below:

- Onboard unit
- Road Side Unit
- Bluetooth
- Wi-Fi
- Mobile Vehicle – 4 wheeler

The project can be carried out to produce the expected outcomes using the following software requirements:

Software requirements:

Operating System	:	Ubuntu 14.01
Simulator	:	NS-3
Packages	:	SUMO, MOVE, Vanet mobilesim, pygtk, pygocanvas, pygraphviz, SDN controller
Languages	:	c++ , python

1.2 Functionality, Working Environment and output format

Network Simulator (Version 3), widely known as NS3, is simply an event driven simulation tool that has proved useful in studying the dynamic nature of communication networks. Simulation of wired as well as wireless network functions and protocols (e.g., routing algorithms, TCP, UDP) can be done using NS2. In general, NS2 provides users with a way of specifying such network protocols and simulating their corresponding behaviors.

The below figure 1.1 shows the basic architecture of NS3. NS3 provides users with a executable command ns which takes on input argument, the name of a Tcl simulation scripting file. Users are feeding the name of a Tcl simulation script (which sets up a simulation) as an input argument of an NS2 executable command ns. In most cases, a simulation trace file is created, and is used to plot graph and/or to create animation.

NS2 consists of two key languages: C++ and Object-oriented Tool Command Language (OTcl). While the C++ defines the internal mechanism (i.e., a backend) of the simulation objects, the OTcl sets u simulation by assembling and configuring the objects as well as scheduling discrete events (i.e., a frontend) and It is advisable to use the C++ objects to set up a simulation using a Tcl simulation script.

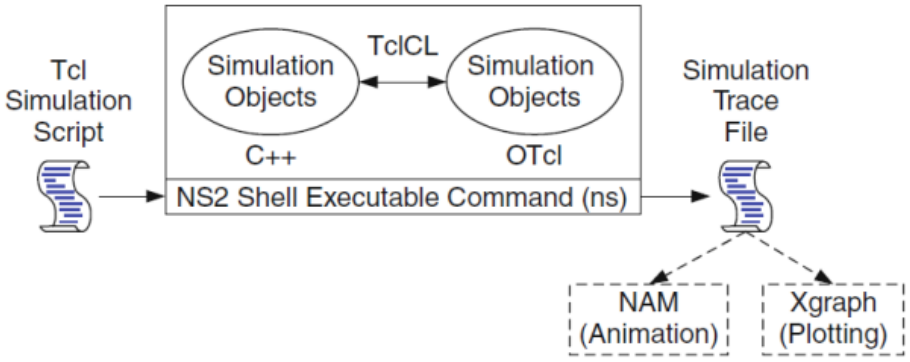

Fig. 1.1: Basic Architecture of NS3

After simulation, NS3 outputs either text-based or animation-based simulation results. To interpret these results graphically and interactively, tools such as NAM (Network AniMator) and XGraph are used. To analyze a particular behavior of the network, users can extract a relevant subset of text-based data and transform it to a more conceivable presentation.

1.2.1 Simulation Design Constraints

- The first step in simulating a network is to design the simulation. In this step, the users should determine the simulation purposes, network configuration and assumptions, the performance measures, and the type of expected results.
- The Second step is Configuring and Running Simulation which includes two phases:
    1. ***Network configuration phase*:** In this phase network components (e.g., node, TCP and UDP) are created and configured according to the simulation design. Also, the events such as data transfer are scheduled to start at a certain time.
    2. ***Simulation Phase*:** This phase starts the simulation which was configured in the Network Configuration Phase. It maintains the simulation clock and executes events chronologically. This phase usually runs until the simulation clock reached a threshold value specified in the Network Configuration Phase. In most cases, it is convenient to define a simulation scenario in a Tcl scripting file (e.g., <file>) and feed the file as an input argument of an NS2 invocation (e.g., executing "ns <file>").
- The third step is the Post Simulation Processing where the main task is to verify the integrity of the program and evaluating the performance of the simulated network. While

the first task is referred to as *debugging*, the second one is achieved by properly collecting and compiling simulation results.

1.2.2 Interface to the interpreter

The following are the different interfaces used in ns2 to implement the project:

- **Class Tcl**

  The class Tcl encapsulates the actual instance of the OTcl interpreter, and provides the methods to access and communicate with that interpreter

- **Class TclObject**

  class TclObject is the base class for most of the other classes in the interpreted and compiled hierarchies. Every object in the class TclObject is created by the user from within the interpreter. An equivalent shadow object is created in the compiled hierarchy

- **Class TclClass**

  This compiled class (class TclClass) is a pure virtual class. Classes derived from this base class provide two functions: construct the interpreted class hierarchy to mirror the compiled class hierarchy; and provide methods to instantiate new TclObjects. Each such derived class is associated with a particular compiled class in the compiled class hierarchy, and can instantiate new objects in the associated class.

- **Class TclCommand**

  This class (class TclCommand) provides just the mechanism for *ns* to export simple commands to the interpreter that can then be executed within a global context by the interpreter.

- **Class EmbeddedTcl**

  ns permits the development of functionality in either compiled code, or through interpreter code, that is evaluated at initialization.

- **Class InstVar**

  This class defines the methods and mechanisms to bind a C++ member variable in the compiled shadow object to a specified OTcl instance variable in the equivalent interpreted object. The binding is set up such that the value of the variable can be set or accessed either from within the interpreter, or from within the compiled code at all times.

1.3 Non Functional Requirements

The non functional requirements considered for simulating the VANET is as below:

1.3.1 Scalability

VANET should be supportive to handle the growing process with hybrid architecture with aggregation technique and p2p technologies to make information exchange more scalable.

1.3.2 Availability

Due to the interaction with vehicular network and physical world developed architecture will be robust enough to withstand unexpected system failure and attacks.

1.3.3 Usability

Growth of VANET used to provide comfort and safety of the passenger.

1.3.4 Security

VANET growing interest and reaches the effort in area of VANET. The project idea offers best service and safety.

1.3.5 Context-Awareness

Protocols should be adaptable to real time environment changes, vehicle density and movement, traffic flow and road topology change. And also the possible consequences the protocol may have on physical world.