APPENDIX 1

SIMULATION TOOL

THE NETWORK SIMULATOR NS-2

Network simulator-2 is used as the simulation tool in this research work. NS was chosen as the simulator, partly because of the range of features it provides and partly because it has an open source code that can be modified and extended. The Research work is implemented in NS-2.31. There is different versions of NS and the latest version is NS-2.34.

Network simulator (NS-2) is a discrete event simulator developed for networking research by the University of California at Berkeley and the VINT project. The Rice Monarch Project formerly known as CMU Monarch Project, has contributed substantial wireless and mobility extensions to the NS-2 network simulator that enable it to accurately simulate mobile nodes connected by wireless network interfaces, including the ability to simulate multi-hop Wireless Ad hoc Networks. Moreover, NS-2 provides important support for simulation of TCP, unicast and multicast routing protocols over wired and wireless (local and satellite) networks. The newest version of NS-2 also provides power consumption simulation support.

Block schematic of a mobile node that implemented in NS-2 is depicted in the Fig shown below. The mobile node makes uses of a routing agent for the purpose of calculating routes to other nodes in the network. When packets are sent from the application agent and are received by the routing agent, the routing agent will decide a path that the packet must travel
through in order to reach its destination and stamps it with this information. It then sends the packet down to the link layer, which uses an Address Resolution Protocol (ARP) to decide the hardware addresses of the neighbouring nodes and translate IP addresses to their correct interfaces.

When this information is known, the packet can be sent down to the interface queue and awaits a signal from the Medium Access Control (MAC) Protocol. When the MAC layer is ready to send the packet to the channel, it fetches the packet form the head of the queue and hands it over to the network interface which in turn sends the packet onto the radio channel. From there this packet is copied and delivered to all network interfaces at the time at which the first bit of the packet would begin arriving at the interface in a physical system, based on the distance between the nodes and the speed of light. Each network interface stamps the packet with the receiving interfaces properties and then invokes the propagation model.

Regarding the packet reception process, the propagation model uses the “transmit” and “receive” stamps to determine the power with which the interface will receive the packet. The receiving network interfaces then use their properties to determine if they have successfully received the packet, and send it up to the MAC layer if appropriate. If the MAC layer receives the packet with no error reported, it sends the packet up to the link layer that passes the packet to the mobile entry point. From here the packet reaches a demultiplexer, which will make a decision on where to send the packet. If the packet has reached its final destination, the “address demux” will pass it to the “port demux”, which will hand the packet to the proper application agent. Otherwise, the address demux shall pass the packet to the default target of the address demux, and the packet shall be forwarded again. Thus, the routing agent at this node will be called on to assign the packet a next hop and pass it back to lower layer. And the procedure will be repeated.
Current version of NS-2 with the CMU Monarch wireless extensions can simulate multi-hop wireless mobile ad hoc networks; however, it provides no support for accurately simulating the physical aspects of multi-hop wireless Village Radio networks or the MAC protocol needed in such environments. In this simulation study, it is made that some modifications to NS-2 to allow accurate simulation of our target mobile wireless networks, i.e. village radio networks. The basic mobile nodes is followed structure under NS-2, but have made certain extensions to NS-2 in physical, MAC and routing layers. With the new elements provided by our extensions to ns, it is possible to construct detailed and accurate simulations of village radio network, a TDMA-based ad hoc network. The following sections will describe more about the important layers implemented in our simulation study.

![SCHEMATIC OF MOBILE NODE UNDER NS-2](image)
A.1.1 PHYSICAL LAYER MODEL

To accurately model the physical layer, NS-2 includes a radio propagation model that supports propagation delay, capture effects, and carrier sensing. The radio propagation model combines both a free space propagation model and a two-ray ground reflection model. When a transmitter is within the reference distance of the receiver, the free space model is used where the signal attenuates as $1/r^2$. Outside of this distance, we use the ground reflection model where the signal falls off as $1/r^4$ (23).

The implementation of the physical layer is based on the design of village radio network where the radio receiver can combine the individual signals to produce a stronger signal instead losing the information due to interference. Therefore VRP does not allow simultaneous transmission of the same packet by multiple users, while neither signal collision nor contention will cause a reception problem in the village radio network. Each mobile node has one or more wireless network interfaces, with all interfaces of the same type (on all mobile nodes) linked together by a single physical channel. When a network interface transmits a packet, it passes the packet to the appropriate physical channel object. This object then computes the propagation delay from the sender to every other interface on the channel and schedules a “packet reception” event for each. This event notifies the receiving interface that the first bit of a new packet has arrived. Last, the packet is passed up to the MAC layer. In order to compute energy consumption of each node, an energy model has been implemented as a node attribute in the network simulator.

The energy model represents level of energy in a mobile host. The energy model in a node has an initial value, which is the level of energy the node has at the beginning of the simulation. It also has a given amount of energy usage for every packet it transmits and receives. The energy of a
mobile host will be decremented for every transmission and reception of packets at the node. To compute the energy of a mobile host, multiply the packet transmission (or reception) time by the transmitting (or receiving) power required by the node's interface or physical layer. When the energy level at the node goes down to zero, no more packets can be received or transmitted by the node.

**Medium Access Control**

The original link layer of NS-2 has implemented the complete IEEE 802.11 standard Medium Access Control (MAC) protocol Distributed Coordination Function (DCF) for mobile ad hoc networks, but NS-2 provides no support for accurately/simulating a multi-hop TDMA-based MAC Protocol needed in our target environments. Thus it has implemented the VRP (Chapter 4) into NS-2 as the Medium Access Control protocol for our simulations of village radio network. According to the multi-hop TDMA-based MAC Protocol VRP, there is no existence of nodes contend for the wireless medium. The transmission of each packet, unicast or broadcast, is scheduled by a TDMA algorithm that assigns the wireless channel for transmission of a data packet to avoid contentions and to reduce the probability of collisions.

An Acknowledgment (ACK) follows each correctly received unicast packet to the sender, which retransmits the packet a limited number of times until this ACK is received. Broadcast packets are not acknowledged by their recipients to reduce energy consumption.

If the MAC layer is idle when an incoming packet is handed up from the network interface, it simply computes the transmission time of the packet and schedules a “packet reception complete” event for itself. When this event occurs, the MAC layer verifies that the packet is error-free, performs destination address filtering, and passes the packet up the protocol stack.
A1.2 ADDRESS RESOLUTION

The Address Resolution Protocol, ARP is implemented in ns-2 to resolve IP addresses to hardware MAC addresses, since the routing protocol VRP operates at the network layer using IP addresses. The address translation by ARP takes place before the packets pass down to the MAC layer.

Interface Queue

The interface queue is implemented for packet buffering. Each node has a queue for packets awaiting transmission by the network interface that holds up to 50 packets and is managed in a drop-tail manner. When a packet comes from the network layer, the link layer will check its next hop address. If the next hop address is an IP address, it needs to be translated to a hardware MAC address by ARP. Once the hardware address of a packet’s next hop is known, the packet is inserted into the interface queue before going to the MAC layer in case the MAC layer is busy. The MAC layer then takes packets from the head of the interface queue and sends them to the network interface when appropriate.

NS (version 2) is an object-oriented, discrete event driven network simulator developed at UC Berkely written in C++ and OTcl. NS is primarily useful for simulating local and wide area networks. Although NS is fairly easy to use once you get to know the simulator, it is quite difficult for a first time user, because there are few user-friendly manuals. Even though there is a lot of documentation written by the developers which has in depth explanation of the simulator, it is written with the depth of a skilled NS user.
The purpose of this research work is to give a new user some basic idea of how the simulator works, how to setup simulation networks, where to look for further information about network components in simulator codes, how to create new network components, etc., mainly by giving simple examples and brief explanations.

**Overview**

NS is an event driven network simulator developed at UC Berkeley that simulates variety of IP networks. It implements network protocols such as TCP and UDP, traffic source behaviour such as FTP, Telnet, Web, CBR and VBR, router queue management mechanism such as Drop Tail, RED and CBQ, routing algorithms such as Dijkstra, and more. NS also implements multicasting and some of the MAC layer protocols for LAN simulations. Currently, NS (version 2) written in C++ and OTcl (Tcl script language with Object-oriented extensions developed at MIT) is available.

![Figure A1.1 Simplified user's view of NS](image)

As shown in Figure A1.1, in a simplified user's view, NS is Object-oriented Tcl (OTcl) script interpreter that has a simulation event scheduler and network component object libraries, and network setup (plumbing)
module libraries (actually, plumbing modules are implemented as member functions of the base simulator object). In other words, to use NS, you program in OTcl script language. To setup and run a simulation network, a user should write an OTcl script that initiates an event scheduler, sets up the network topology using the network objects and the plumbing functions in the library, and tells traffic sources when to start and stop transmitting packets through the event scheduler. The term "plumbing" is used for a network setup, because setting up a network is plumbing possible data paths among network objects by setting the "neighbour" pointer of an object to the address of an appropriate object. When a user wants to make a new network object, he or she can easily make an object either by writing a new object or by making a compound object from the object library, and plumb the data path through the object. This may sound like complicated job, but the plumbing OTcl modules actually make the job very easy. The power of NS comes from this plumbing.

Another major component of NS beside network objects is the event scheduler. An event in NS is a packet ID that is unique for a packet with scheduled time and the pointer to an object that handles the event. In NS, an event scheduler keeps track of simulation time and fires all the events in the event queue scheduled for the current time by invoking appropriate network components, which usually are the ones who issued the events, and let them do the appropriate action associated with packet pointed by the event.

Network components communicate with one another passing packet; however this does not consume actual simulation time. All the network components that need to spend some simulation time handling a packet (i.e. need a delay) use the event scheduler by issuing an event for the packet and waiting for the event to be fired to itself before doing further action handling the packet. For example, a network switch component that
simulates a switch with 20 microseconds of switching delay issues an event for a packet to be switched to the scheduler as an event 20 microsecond later. The scheduler after 20 micro second de-queues the event and fires it to the switch component, which then passes the packet to an appropriate output link component.

Another use of an event scheduler is timer. For example, TCP needs a timer to keep track of a packet transmission time out for retransmission (transmission of a packet with the same TCP packet number but different NS packet ID). Timers use event schedulers in a similar manner that delay does. The only difference is that timer measures a time value associated with a packet and does an appropriate action related to that packet after a certain time goes by, and does not simulate a delay.

NS is written not only in OTcl but in C++ also. For efficiency reason, NS separates the data path implementation from control path implementations. In order to reduce packet and event processing time (not simulation time), the event scheduler and the basic network component objects in the data path are written and compiled using C++. These compiled objects are made available to the OTcl interpreter through an OTcl linkage that creates a matching OTcl object for each of the C++ objects and makes the control functions and the configurable variables specified by the C++ object act as member functions and member variables of the corresponding OTcl object. In this way, the controls of the C++ objects are given to OTcl. It is also possible to add member functions and variables to a C++ linked OTcl object. The objects in C++ that do not need to be controlled in a simulation or internally used by another object do not need to be linked to OTcl. Likewise, an object (not in the data path) can be entirely implemented in OTcl. Figure A1.2 shows an object hierarchy example in C++ and OTcl. One thing to note in the figure is that for C++ objects that have an OTcl linkage forming
a hierarchy, there is a matching OTcl object hierarchy very similar to that of C++.

![Figure A1.2 C++ and OTCL: the duality](image)

![Figure A1.3 Architectural view of NS](image)

Figure A1.3 shows the general architecture of NS. In this figure a general user (not an NS developer) can be thought of standing at the left bottom corner, designing and running simulations in Tcl using the simulator objects in the OTcl library.

The event schedulers and most of the network components are implemented in C++ and available to OTcl through an OTcl linkage that is
implemented using tcl. The whole thing together makes NS, which is a OO extended Tcl interpreter with network simulator libraries.

This section briefly examined the general structure and architecture of NS. At this point, one might be wondering about how to obtain NS simulation results. As shown in Figure A1.1, when a simulation is finished, NS produces one or more text-based output files that contain detailed simulation data, if specified to do so in the input Tcl (or more specifically, OTcl) script. The data can be used for simulation analysis (two simulation result analysis examples are presented in later sections) or as an input to a graphical simulation display tool called Network Animator (NAM) that is developed as a part of VINT project. NAM has a nice graphical user interface similar to that of a CD player (play, fast forward, rewind, pause and so on), and also has a display speed controller. Furthermore, it can graphically present information such as throughput and number of packet drops at each link, although the graphical information cannot be used for accurate simulation analysis.

**OTcl: The User Language**

As mentioned in the overview section, NS is basically an OTcl interpreter with network simulation object libraries. It is very useful to know how to program in OTcl to use NS. This section shows an example Tcl and OTcl script, from which one can get the basic idea of programming in OTcl. These examples are from the 5th VINT/NS Simulation Tutorial/Workshop. This section and the sections after assumes that the reader installed NS, and is familiar with C and C++. 
Example 1 is a general Tcl script that shows how to create a procedure and call it, how to assign values to variables, and how to make a loop. Knowing that OTcl is Object oriented extension of Tcl, it is obvious that all Tcl commands work on OTcl - the relationship between Tcl and OTcl is just same as C and C++. To run this script you should download ex-tcl.tcl, and type "ns ex-tcl.tcl" at your shell prompt - the command "ns" starts the NS (an OTcl interpreter). You will also get the same results if you type "tcl ex-tcl.tcl", if tcl8.0 is installed in your machine.

```
# Writing a procedure called "test"
proc test () {
  set a 43
  set b 27
  set c [expr $a + $b]
  set d [expr $a - $b] 1 $c]
  for {set k 0} {$k < 10} {incr k} {
    if ($k < 5) {
      puts "$k < 5, pow = [expr pow($d, $k)]"
    } else {
      puts "$k >= 5, mod = [expr $d % $k]"
    }
  }
}

# Calling the "test" procedure created above
test
```

Example 1. A Sample Tcl Script

In Tcl, the keyword proc is used to define a procedure, followed by an procedure name and arguments in curly brackets. The keyword set is used to assign a value to a variable. [expr ...] is to make the interpreter calculate the value of expression within the bracket after the keyword. One thing to
The next example is an object-oriented programming example in OTcl. This example is very simple, but shows the way which an object is created and used in OTcl. As an ordinary NS user, the chances that you will write your own object might be rare. However, since all of the NS objects that you will use in a NS simulation programming, whether or not they are written in C++ and made available to OTcl via the linkage or written only in OTcl, are essentially OTcl objects, understanding OTcl object is helpful.

Example 2 is an OTcl script that defines two object classes, "mom" and "kid", where "kid" is the child class of "mom", and a member function called "greet" for each class. After the class definitions, each object instance is declared, the "age" variable of each instance is set to 45 (for mom) and 15 (for kid), and the "greet" member function of each object instance is called.
```tcl
# add a member function call "greet"
Class mom
mom instproc greet () {
    $self instvar age_
    puts "$age_ year old mom say:
          How are you doing?"
}

# Create a child class of "mom" called "kid"
# and override the member function "greet"
Class kid -superclass mom
kid instproc greet () {
    $self instvar age_
    puts "$age_ year old kid say:
          What's up, dude?"
}

# Create a mom and a kid object, set each age
set a [new mom]
$a set age_ 45
set b [new kid]
$b set age_ 15

# Calling member function "greet" of each object
$a greet
$b greet
```

Example 2 A Sample OTcl Script

The keyword **Class** is to create an object class and **instproc** is to define a member function to an object class. Class inheritance is specified using the keyword **-superclass**. In defining member functions, **$self** acts same as the "this" pointer in C++, and **instvar** checks if the following variable name is already declared in its class or in its superclass.

If the variable name given is already declared, the variable is referenced, if not a new one is declared. Finally, to create an object instance, the keyword **new** is used as shown in the example. Downloading ex-otcl.tcl and executing "ns ex-otcl.tcl" will give you the following result:
Event Scheduler

This section talks about the discrete event schedulers of NS. As described in the Overview section, the main users of an event scheduler are network components that simulate packet-handling delay or that need timers. Figure A1.4 shows each network object using an event scheduler. Note that a network object that issues an event is the one who handles the event later at scheduled time. Also note that the data path between network objects is different from the event path. Actually, packets are handed from one network object to another using send (Packet* p) {target_->recv(p)}; method of the sender and recv(Packet*, Handler* h = 0) method of the receiver.

Figure A1.4 Discrete event scheduler
NS has two different types of event schedulers implemented. These are real-time and non-real-time schedulers. For a non-real-time scheduler, three implementations (List, Heap and Calendar) are available; even though they are all logically perform the same. This is because of backward compatibility: some early implementation of network components added by a user (not the original ones included in a package) may use a specific type of scheduler not through public functions but hacking around the internals. The Calendar non-real-time scheduler is set as the default. The real-time scheduler is for emulation, which allows the simulator to interact with a real network. Currently, emulation is under development although an experimental version is available. The following is an example of selecting a specific event scheduler:

```tcl
set ns [new Simulator]
$ns use-scheduler Heap
```

Another use of an event scheduler is to schedule simulation events, such as when to start an FTP application, when to finish a simulation, or for simulation scenario generation prior to a simulation run. An event scheduler object itself has simulation scheduling member functions such as at time "string" that issue a special event called At Event at a specified simulation time. An "AtEvent" is actually a child class of "Event", which has an additional variable to hold the given string. However, it is treated the same as a normal (packet related) event within the event scheduler. When a simulation is started, and as the scheduled time for an AtEvent in the event queue comes, the AtEvent is passed to an "AtEvent handler" that is created once and handles all AtEvents, and the OTcl command specified by the string
field of the AtEvent is executed. The following is a simulation event scheduling line added version of the above example.

```tcl
set ns [new Simulator]
$ns use-scheduler Heap
$ns at 300.5 "complete_sim"
```

```tcl
proc complete_sim {} {
}
```

```tcl
Simulator instproc now # return scheduler's notion of current time
Simulator instproc at args # schedule execution of code at specified time
Simulator instproc at-now args # schedule execution of code at now
Simulator instproc after n args # schedule execution of code after n secs
Simulator instproc run args # start scheduler
Simulator instproc halt # stop (pause) scheduler
```

**Network Components**

This section talks about the NS components, mostly compound network components. Figure 5 shows a partial OTcl class hierarchy of NS, which will help understanding the basic network components. The root of the hierarchy is the TclObject class that is the superclass of all OTcl library objects (scheduler, network components, timers and the other objects
including NAM related ones). As an ancestor class of TclObject, NsObject class is the superclass of all basic network component objects that handle packets, which may compose compound network objects such as nodes and links. The basic network components are further divided into two subclasses, Connector and Classifier, based on the number of the possible output data paths.

![Class hierarchy diagram](image)

**Figure A1.5 Class hierarchy (PARTIAL)**

The basic network objects that have only one output data path are under the Connector class, and switching objects that have possible multiple output data paths are under the Classifier class.

**Node and Routing**

A node is a compound object composed of a node entry object and classifiers as shown in Figure A1.6. There are two types of nodes in NS. A unicast node has an address classifier that does unicast routing and a port classifier. A multicast node, in addition, has a classifier that classify multicast packets from unicast packets and a multicast classifier that performs multicast routing.
In NS, Unicast nodes are the default nodes. To create Multicast nodes the user must explicitly notify in the input OTcl script, right after creating a scheduler object, that all the nodes that will be created are multicast nodes. After specifying the node type, the user can also select a specific routing protocol other than using a default one.

**Link**

A link is another major compound object in NS. When a user creates a link using a duplex-link member function of a Simulator object, two simplex links in both directions are created as shown in Figure A1.7.
One thing to note is that an output queue of a node is actually implemented as a part of simplex link object. Packets de-queued from a queue are passed to the Delay object that simulates the link delay, and packets dropped at a queue are sent to a Null Agent and are freed there. Finally, the TTL object calculates Time to live parameters for each packet received and updates the TTL field of the packet.

- **Tracing**

In NS, network activities are traced around simplex links. If the simulator is directed to trace network activities (specified using $ns trace-all file or $ns namtrace-all file), the links created after the command will have the following trace objects inserted as shown in Figure 8. Users can also specifically create a trace object of type between the given src and dst nodes using the create-trace {type file src dst} command.

![Figure A1.8 Inserting trace objects](image)

When each inserted trace object (i.e. EnqT, DeqT, DrpT and RecvT) receives a packet, it writes to the specified trace file without consuming any simulation time, and passes the packet to the next network object. The trace format will be examined in the General Analysis Example section.
Queue Monitor

Basically, tracing objects are designed to record packet arrival time at which they are located. Although a user gets enough information from the trace, he or she might be interested in what is going on inside a specific output queue. For example, a user interested in RED queue behaviour may want to measure the dynamics of average queue size and current queue size of a specific RED queue (i.e. need for queue monitoring). Queue monitoring can be achieved using queue monitor objects and snoop queue objects as shown in Figure 9.

![Link with Snoop Queue Objects](image)

**Figure A1.9 Monitoring Queue**

When a packet arrives, a snoop queue object notifies the queue monitor object of this event. The queue monitor using this information monitors the queue. A RED queue monitoring example is shown in the RED Queue Monitor Example section. Note that snoop queue objects can be used in parallel with tracing objects even though it is not shown in the above figure.
Packet Flow Example

Until now, the two most important network components (node and link) were examined. Figure A1.10 shows internals of an example simulation network setup and packet flow. The network consists of two nodes (n0 and n1) of which the network addresses are 0 and 1 respectively. A TCP agent attached to n0 using port 0 communicates with a TCP sink object attached to n1 port 0. Finally, an FTP application (or traffic source) is attached to the TCP agent, asking to send some amount of data.

![Packet Flow Example](image)

**Figure A1.10 Packet flow example**

Note that the above figure does not show the exact behaviour of a FTP over TCP. It only shows the detailed internals of simulation network setup and a packet flow.

**NS can simulate the following**

1. **Topology**  : Wired, wireless

2. **Scheduling Algorithms**: RED, Drop Tail,
3. Transport Protocols : TCP, UDP

4. Routing : Static and dynamic routing

5. Application : FTP, HTTP, Telnet, Traffic generators

Mobile Networking in NS

The wireless model essentially consists of the Mobile Node at the core with additional supporting features that allows simulations of multi-hop ad-hoc networks, wireless LANs etc. The Mobile Node object is a split object. The C++ class Mobile Node is derived from parent class Node. A Mobile Node thus is the basic Node object with added functionalities of a wireless and mobile node like ability to move within a given topology, ability to receive and transmit signals to and from a wireless channel etc. A major difference between them is that a mobile Node is not connected by means of Links to other nodes or mobile nodes. Mobile Node is the basic ns Node object with added functionalities like movement, ability to transmit and receive on a channel that allows it to be used to create mobile, wireless simulation environments. The class Mobile Node is derived from the base class Node. The four ad-hoc routing protocols that are currently supported are Destination Sequence Distance Vector (DSDV), Dynamic Source Routing (DSR), Temporally ordered Routing Algorithm (TORA) and Adhoc On-demand Distance Vector (AODV).

Network Animator (NAM)

Network animator (NAM) is an animation tool for viewing network simulation traces and real world packet teaches. It supports topology layout, packet level animation and various DATA inspection tools. Before starting to
use NAM, trace file need to be created. This trace file is usually generated by NS. It contains topology information, e.g., nodes and links, as well as packet traces. During a simulation, the user can produce topology configurations, layout information and packet traces using tracing events in NS. Once the trace file is generated, NAM can be used to animate it. Upon startup, NAM will read the trace file, create topology, pop up a window, do layout if necessary and then pause at time 0. Through its user interface, NAM provides control over many aspects of animation.

In Figure A1.11 a screenshot of a NAM window is shown, where the most important function is explained. Although the NAM software contains bugs, as do the NS software, it works fine most of the times and causes only little trouble. NAM is an excellent first step to check that the scenario works as expected. NS and NAM can also be used together for educational purpose and to easily demonstrate different networking issues. In this thesis NS-2 is used to simulate and test the various approaches and NAM is used to produce the graphical output.
Figure A1.11 Screenshot of a NAM window

NETWORK SIMULATOR 3

NS-3 is a discrete-event network simulator in which the simulation core and models are implemented in C++. NS-3 is built as a library which may be statically or dynamically linked to a C++ main program that defines the simulation topology and starts the simulator. NS-3 also exports nearly its entire API to Python, allowing Python programs to import an “NS3” module in much the same way as the NS-3 library is linked by executables in C++. 
The source code for NS-3 is mostly organized in the src directory and can be described by the diagram in Software organization of NS-3. In general, modules only have dependencies on modules beneath them in the figure. The core of the simulator is described; those components that are common across all protocol, hardware, and environmental models. The simulation core is implemented in src/core. Packets are fundamental objects in a network simulator and are implemented in src/network. These two simulation modules by themselves are intended to comprise a generic simulation core that can be used by different kinds of networks, not just Internet-based networks.

The above modules of NS-3 are independent of specific network and device models, which are covered in subsequent parts of this manual. In addition to the above NS-3 core, two other modules that supplement the core C++-based API. NS-3 programs may access the entire API directly or may make use of a so-called helper API that provides convenient wrappers or encapsulation of low-level API calls. The fact that NS-3 programs can be
written to two APIs (or a combination thereof) is a fundamental aspect of the simulator. Python is supported in NS-3 before moving onto specific models of relevance to network simulation.

**Overview of NS-3 random numbers**

NS-3 random numbers are provided via instances of NS-3 Random Variable.

- By default, NS-3 simulations use a fixed seed; if there is any randomness in the simulation, each run of the program will yield identical results unless the seed and/or run number is changed.

- In ns-3.3 and earlier, NS-3 simulations used a random seed by default; this marks a change in policy starting with NS-3.4.

- To obtain randomness across multiple simulation runs, you must either set the seed differently or set the run number differently. To set a seed, call `ns3::SeedManager::SetSeed()` at the beginning of the program; to set a run number with the same seed, call `ns3::SeedManager::SetRun()` at the beginning of the program.

- Each Random variable used in NS-3 has a virtual random number generator associated with it.