CHAPTER 4

SIMULATION ENVIRONMENTS

One requirement of this thesis is the use of a network simulator to evaluate the performance of the fuzzy logic control added. This chapter therefore describes the evaluation of several network simulators for use in this thesis. Both commercial and free types of simulators were considered to determine the one best suited for the task.

Fundamental basics such as the acquisition, installation and running of the reviewed software is not included in the text of this report. There is more than adequate information available about how to perform these tasks through manuals and online documentation and repeating them here would be redundant.

4.1 Network Simulations

Network simulations are model representations of real network systems that are designed for the purpose of conducting experiments to understand the behaviour of the system and/or evaluate different strategies for the operation of the system (Breslau et al. 2000). Any proper simulation should behave or operate like the system it was designed after when given a set of controlled inputs so as to obtain an accurate depiction of the actual working system. Thus it is important that a simulation handles

all the actual processes that occur, both internally and externally, and not just directly simulate the results by means of calculations.

Simulations of networks are often performed in place of constructing an actual network. To physically build a network just for testing purposes is unjustified as it is usually impractical and more often than not, is very expensive. In some cases, it is just simply impossible to construct the network especially when it involves changing hardware specifications which could require special equipment. Running simulations allows for more precise gathering and measurement of simulation data since every aspect of the simulation can be tracked as variables whilst in a real network it would involve various collection devices and additional software. The use of simulation therefore offers a practical and more precise means of obtaining accurate information that contributes to the planning and designing of a new system.

4.1.1. Modelling a Network

Simulations of networks begin with the construction of the simulation models. A simulation model attempts to emulate the functional components in an actual network – nodes, protocols, resources and links – by replicating the components' characteristics and tasks it normally performs. These network components can then communicate with each other through model defined or user provided interfaces to form the entire network topology. Since most models are an abstraction of real systems, it cannot fully represent any system in whole but must implement elements that are vital to the simulation objective.

The following categories of models are commonly found in network simulators although not all the examples of models given in each category are included.

Table 4.1 – Common simulation models

Model type	Examples
Application layer	FTP, HTTP, VoIP, Media streaming, CBR, VBR
Data link layer	ATM, Ethernet (IEEE 802.3X), Frame relay, VLAN, Token ring, WLAN (IEEE 802.11x), Satellite
Network layer protocol	IP, IPX, RSVP
Routing	OSPF, BGP, IGRP, EIGRP, RIP, MPLS
Transport layer	TCP, UDP
Physical layer	ISDN, xDSL, Radio
Node/Station	Mobile, Base station, AP
Propagation	Two-ray, Freespace, Shadowing
Topology	Internet, Grid, Random waypoint

4.1.2 Approaches to Simulation

There are many approaches to running simulations but most network simulators typically use one of two simulation methods: discrete-event or analytical. In analytical simulations, mathematical equations are used to predict network and application performance in the system. Once the formulas and equations are derived (e.g. queuing theory, probability theory, etc.), the evaluation of the system is very quick but the accuracy of the simulation is sacrificed because analytical simulations have a habit of making assumptions and simplifications of the simulated model. One problem with analytical simulations is that while it can simulate small sized networks

exactly and quickly, it does not scale well to large networks (inaccuracies are magnified) and cannot simulate complex network configurations that might be needed.

On the other hand, discrete-event simulators create an extremely detailed, packet-by-packet model of predicted network activity by tracing events that occur at specific finite points in simulation time (not real time). The task of tracing each and every event that occurs for all objects in the model often requires extensive calculations and processing to simulate a brief period of "real time". Typical discrete-event simulations are painfully slow and can take anywhere from minutes to days just to run to completion even though the actual simulated amount of time is only a fraction of the time taken. Discrete-event simulators also have heavy demands on processing power and usually require extremely large amounts of memory to run successfully. It is unsurprising to hear about discrete-event simulations requiring many gigabytes of memory just to run as the amount required often grows exponentially with the addition of node, objects or events to the simulation model.

To provide the best of two worlds, some have taken the approach of hybrid simulation into their simulator software. In this case, discrete-event and analytical simulation techniques are combined to provide detailed network simulation whilst still running quickly. Some simulators allow control over the detail generated by the simulator: turning up the level of detail increases run-time proportionally but scale it back and run-time is shortened.

4.2 Existing Network Simulators

The market for network simulators is quite wide and there are many simulators available, each with their own features, advantages and disadvantages. Some simulators are purpose-built and are very focused on a particular area of research and can handle only a particular type of network but can handle it very well. Then there are the general-purpose simulators that support a multitude of protocols, standards and models thus making it suitable for simulating different types of networks. One common feature between the simulators is that they base their simulation on the discrete-event technique, although some do implement certain hybrid techniques for better performance. The following section evaluates a few simulators available.

4.2.1 GloMoSim/Qualnet

GloMoSim is a scalable network simulator for wireless networks developed by the UCLA Parallel Computing Library department. GloMoSim, which stands for *Global Mobile Information Systems Simulation Library,* is built on UCLA's own parallel simulation language called PARSEC which itself in written in C. The basic node model in GloMoSim follows the 7-layered OSI architecture with standard application interfaces (APIs) to be used between layers. This arrangement allows users to add models to a specific layer without having to modify the rest (UCLA 2001).

GloMoSim runs simulations based on scripts that define the components in the simulated network. It does not seem to offer any form of a graphical interface to handle simulations but most other simulators do not either. This simulator is developed specifically for mobile networks (hence the name sake) and does not

provide much support for wired networks (as of the latest version). It also runs only on UNIX-based machines.

GloMoSim v2.0 is freeware and downloadable by educational institutions that apply for use of the software in research but this software has not been updated since its last version released in year 2000. A commercial product derived from GloMoSim called QualNet has since been developed that claims to be the fastest network simulator available but this software requires a purchased license to install and operate (although a 2-week evaluation version is available)

4.2.2 OPNET

OPNET is a commercial general purpose simulation program for modelling communication systems including mobile, wired and wireless networks. Originally developed at MIT in 1987 and touted as the first ever commercial simulator, it is now being handled and further developed by OPNET Technologies. It is written in object-oriented C/C++ using the hybrid simulation approach and offers a graphical interface for running simulations in either Unix (Solaris, HP-UX) or Windows machines.

OPNET has the advantage of being able to offer a wide range of simulation models for various protocols and models including vendor specific simulation models for products from companies like CISCO, 3COM, Fore, Lucent, HP and Intel. The included library seems to be the most complete and comprehensive set available out of any of the simulators available in the market. The included documentation is also very complete and covers most aspects of running, adapting and adding to the simulator.

On the other hand, OPNET is an expensive piece of software and this is mainly due to the licensing requirements met to include vendor specific products. A free research version is available to those that qualify but the activation license timeout is in three months from date of application (although it is renewable). Renewing the license to continue using it is a hassle.

4.2.3 cnet

The 'cnet' network simulator is developed at the School of Computer Science and Software Engineering, University of Western Australia. Written in ANSI-C with Tcl/Tk as its graphical front-end, cnet enables experimentation with various data-link, network and transport layer networking protocols whilst providing a basic application and physical layer (that also can be changed). Although it is a simulator, its main use is in class work assignments in which this software is used to allow undergraduate students to develop and test out their own protocols.

Similar to GloMoSim, cnet runs only in Unix based environments but has the advantage of being able to offer a pretty comprehensive user interface in XWindows. Another advantage is the small size of the program with the entire source tar package weighing in at around 1.1MB only (version 2.0.9 dated 13th May 2004) so the demands on disk space are small. Documentation is rather sparse though mainly because the use of the software is taught in the university lectures and a complete manual is not available.

4.2.4 JavaSim

JavaSim is a discrete-event process-based simulator developed by the Department of Computer Science, University of Newcastle upon Tyne UK. It is written using Java and based on the university's own proprietary object-oriented simulation package called C++SIM which was developed using C++.

Running in either Windows or Unix machines, JavaSim is easily the smallest simulator package here with the source code package of 52.5kB only although installation of the software still requires the Java compiler/runtime and a C/C++ compiler. There is no interface and simulations are run based on calls to the respective Java functions built into the classes that are inherited from. The main disadvantage is that JavaSim is in its infancy and does not support many functions yet. Documentation is also sparse making it hard to understand the coding required in the software.

4.2.5 NIST ATM/HFC Network Simulator

The NIST Simulator is discrete-event purpose built simulator meant for studying and evaluating ATM (Asynchronous Transfer Mode) and HFC (Hybrid Fibre Coaxial) networks. It is developed by the National Institute of Standards and Technology (NIST) in the USA as a free tool to perform network planning and protocol analysis of ATM/HFC networks without the expense of building a real network (Golmie et al. 1998).

The NIST simulator only runs in Unix in the XWindows environments and provides a graphical user interface to design and execute a simulation. The package is written

in C with additional coding for the X environment. Unfortunately this simulator does not meet the needs of this project as it is developed only for the specific type of ATM/HFC network and does not support any other type.

4.2.6 NS

NS (which unsurprisingly stands for Network Simulator) is a simulator developed by the Information Sciences Institute (ISI), University of Southern California (VINT 2003). Targeted mainly at networking research, NS is a discrete event general purpose simulator with support for various networking protocols, topologies and technologies. Throughout the years, the simulator has been through two major revisions known simply as NSv1 and NSv2. Whilst further development in NSv1 has stopped, NSv2 still continues to advance by incorporating new features and supporting more networking technologies. Research funding for NS is currently provided by DARPA (with SAMAN) and NSF (with CONSER) with contributions from other researchers including those from the UCB Daedelus and CMU Monarch projects and Sun Microsystems.

Running mainly on Unix based machines (works in Windows with Cygwin), NS also does not have any form of a user interface built-in. Simulations are run based on TCL scripts that can be written quickly or adapted from available scripts. Several 'helper' applications have been contributed by other developers to assist in using NS and these include a graph plotter and analyser, topology generator and simulation animator. There are also plugins available to port data to and from other programs such as MATLAB and gnuplot.

NS has an advantage of being freeware and is very modular in programming style plus it supports many different protocols and models in the default package. Those not included are usually available as add-on patches that incorporate the functionality into NS once installed. Support and documentation is also abundant since many research projects have been successfully performed using NS. The disadvantage of NS is that it requires a strong background in C programming and also understanding in TCL programming. The modularity of the code makes adding features both a blessing and a chore since it is hard to locate the specific file to alter amongst the thousands in the package. Once the particular file is found though, adding features is easy and there is no need to worry about other parts of the program as long as the added function/feature conforms to the programming style to ensure interoperability.

4.2.7 Summary

The simulators described above only represent a tiny proportion of the available simulators in the market but are the ones regularly used in research and development purposes and are easily obtainable. Each simulator was evaluated based on the requirements of this project in terms of functionality, programming language use, support and code documentation. Features such as graphical interfaces and operating system platform compatibility were not taken into account as it does not matter much. Several other simulators were also evaluated including AdventNet but problems with installation and licensing cause it not to be included in this report.

The following table gives a side-by-side comparison of the simulators mentioned above in terms of programming language used, documentation level, modularity of code and model support.

Table 4.2 – Summary of reviewed network simulators

Simulator	License type	Language	Documentation	Code modularity	Simulation model support
GloMoSim	Free	C/PARSEC	OK	Yes	Wireless only
OPNET	Commercial	C/C++	Good	Yes	Good
cnet	Free	C/ Tcl/Tk	OK	Yes	Poor
JavaSim	Free	Java	OK	No	Poor
NIST ATM/HFC	Free	C	Poor	No	ATM/HFC only
NS	Free	C/OTcl	Good	Yes	Good

For this project, the NS simulator was chosen as the simulator mainly because of the abundance of documentation available to refer to. The programmers help forums also remain active with plenty of help and support available. Prior experience with the simulator also contributed to the choosing of NS as the simulator here. The simulator is also open source making the source code available entirely free for non-commercial purposes.

The next section describes the features of the NS simulator version 2.19b which was used in this thesis and the modules related to the work done. Note that "*NS*" actually refers to the NS v2 of the simulator package unlike other research efforts that prefer to use either the term "*NS-2*", "*ns2*" or other variations. This style mirrors how it is referred to in the documentation and write-ups provided by the developers at ISI.

4.3 Mobile Networking Simulations in NS

The wireless model in NS was originally offered as an extension to the default package by the CMU Monarch Group (CMU 1999). As of versions 2.18 and up of the simulator, the model has been ported over and included as part of the default distribution package. Changes to the original CMU code were made by contributors and developers at ISI to ensure interoperability with other existing modules in NS.

The default wireless model in NS basically revolves around the mobile node with additional features to allow simulations of multi-hop ad-hoc networks, wireless LANs and such (VINT 2003). The mobile node class (*MobileNode.{cc,h}*) is derived from the parent class *Node* and thus inherits all the properties of a basic node. It differs from the basic node through added functionalities such as node movement, position logging and transmission/receiving operations to work with the wireless medium in wireless simulations.

4.3.1 The MobileNode Structure and Organization

The following figure 4.1 shows a logical representation of how the mobile nodes are connected together in an NS simulation. Each node is a separate and independent object, responsible for computing its own position, movement and velocity over time. Each node can have multiple *network interfaces* that are connected to the *channel*. The channel functions as conduits to distribute packets to every mobile node connected to it and actually represents the particular frequency, modulation and coding scheme used for transmission.

There is no actual physical pipeline as shown in the figure (shown as the channel) to which the nodes connect to, instead the nodes use a *radio propagation model* to transmit and receive packets over the air using radio waves.

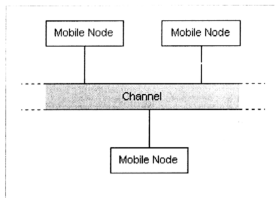

Figure 4.1 – Representation of connections between nodes and the channel

As with all objects in NS, the MobileNode object is split into its C++ and OTcl parts. Mobility features are implemented using C++ whilst the creation and plumbing of the network stack is done using OTcl. To create a mobile node the following procedures are called in the simulation script while passing appropriate values.

```
$ns_ node_config  -adhocRouting $opt(adhoc) \
                  -llType $opt(ll) \
                  -macType $opt(mac) \
                  -ifqType $opt(ifq) \
                  -ifqLen $opt(ifqlen) \
                  -antType $opt(ant) \
                  -propType $opt(prop) \
                  -phyType $opt(netif) \
                  -channel [new $opt(chan)] \
                  -topoInstance $topo \
                  -agentTrace ON \
```

```
                -routerTrace ON \
                -macTrace ON \
                -movementTrace OFF
    set node_(0) [$ns_ node]
```

This creates a single basic mobile node that now looks like the diagram below. This is the node structure used in this thesis.

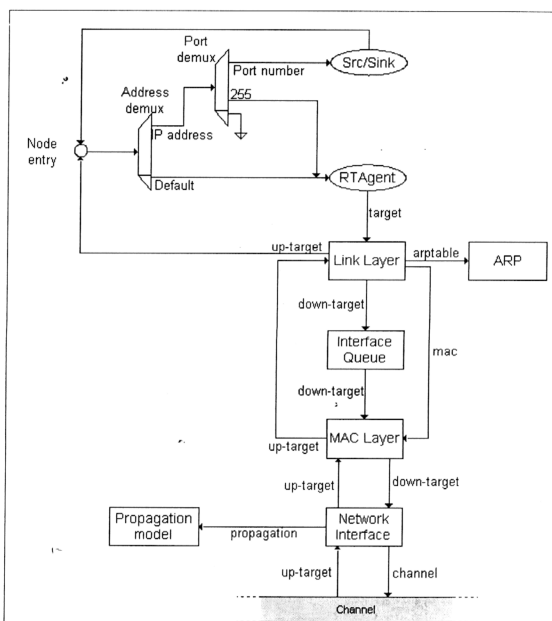

Figure 4.2 – Mobile node structure in NS

A new mobile node in NS is defined in the file *mobilenode.cc* and is created and plumbed together using OTcl in *mobilenode.h*. When a new node is created in OTcl, the network stack for the node is set up as below

```
# Local variables
  set nullAgent_ [$ns_ set nullAgent_]
  set netif $netif_($t)
  set mac $mac_($t)
  set ifq $ifq_($t)
  set ll $ll_($t)
# Initialise ARP table only once
  if ($arptable_ == "") {
      set arptable_ [new ARPTable $self $mac]
      set drpT [cmu-trace Drop "IFQ" $self]
      $arptable_ drop-target $drpT
  }
# Link Layer
  $ll arptable $arptable_
  $ll mac $mac
  $ll up-target [$self entry]
  $ll down-target $ifq
# Interface Queue
  $ifq target $mac
  $ifq set qlim_ $qlen
  set drpT [cmu-trace Drop "IFQ" $self]
  $ifq drop-target $drpT
# Mac Layer
  $mac netif $netif
  $mac up-target $ll
  $mac down-target $netif
```

```
    $mac nodes $opt(nn)
  # Network Interface
    $netif channel $channel
    $netif up-target $mac
    $netif propagation $pmodel
    $netif node $self            ;# bind node <---> interface
    $netif antenna $ant_($t)     ;# attach antenna
  # Physical Channel
    $channel add $netif    ;# add to list of interfaces
```

From the code fragment above, *$opt(xx)* variables set in the *node_config* procedure on the previous page are used to create a null agent, network interface, MAC, link layer and interface queue. These components are connected either as a down-target or up-target of each other. It is clear that the layer above the MAC is the link layer as stated in the *up-target* command whilst the *down-target* is the network interface. Each network interface has a channel, propagation type and an up-target of the MAC layer. The present node (the one being created) is added to the network interface with the command *node* and an antenna is attached. Finally, the interface is added to the channel such as in figure 4.1.

4.3.2 Outgoing and Incoming Packets

During simulations, the path taken through the layers depend on the type of packet being sent or received. Packets sent by a source agent on the mobile node are handed to the entry point of the node, which passes them to the address demultiplexer. The destination address of the packet is obtained and, if it matches the node's own address, passes the packet up to the port demultiplexer to deliver it to the right application. If the destination is another node, they are handed down to the routing

protocol. The routing protocol sets the *next_hop* field in the packet's common header to the address of the next node the packet should be sent to. If the next hop address is an IP address, the link layer object then proceeds to query the ARP object to obtain the hardware address. If the ARP does not have any information about the next hop, the packet is queued and an ARP request packet is sent using a broadcast address to perform address resolution. Once the hardware address is known, the packet is set and inserted into the network interface queue (IFQ). The MAC object removes packets from the IFQ head and sends them onto the channel using the interface when appropriate. In the network interface, the packet is stamped with properties such as the transmission power and node location, and then puts the packet onto the channel when a copy is delivered to all the nodes on the channel.

The receiving process occurs when copies of the sent packet arrives at a node after an allotted time (after propagation time controlled by the scheduler). Upon receipt, the network interface stamps the packet header with the receiving node's properties and invokes the propagation model. The propagation model uses the transmit and receive header stamps plus the properties of the receiving node's interface to determine the power with which the interface will receive the packet. If adequate, the packet is successfully captured and handed to the MAC layer. At the MAC layer, if the packet is collision and error free, the packet is passed to the node's entry point. The packet is demultiplexed and checked to see if it has arrived at the destination node which will hand the packet to the proper sink agent at the specific port. If not, the packet is sent to the default target address and the routing agent called to assign the next hop for the packet before passing it back to the link layer.

4.3.3 Ad-hoc Routing Models in NS

The default NS package includes four ad-hoc routing protocols : DSDV, DSV, AODV and TORA. Of these four, DSR, AODV and TORA are on-demand protocols whilst DSDV is table driven. In table driven routing, consistent and up-to-date routing information is maintained at all nodes whereas in on-demand routing, routes are only created when required by the source node (Misra 2000). This section briefly explains how each protocol works.

Dynamic Destination-Sequenced Distance-Vector Routing Protocol (DSDV)

In this routing protocol, every mobile node maintains a routing table that lists all available destinations and the routes to get there (number of hops and sequence number). The nodes periodically transmit their routing tables to their immediate neighbours to update the routing information. These updates may also be triggered by any significant change in the network such as a node disappearing and bringing down a link. The updates themselves can be one of two types – a full dump which sends the entire routing table or an incremental update that sends only routing table entries that have changed since the last update.

The advantage of this protocol is that a node immediately knows the route to a destination by referring to its routing table and does not need to wait while the node sends out a route request. The disadvantage with this protocol is the task of keeping the routing table up-to-date. In a fast changing network, updates can be sent so often that it would take up the precious available bandwidth. There is also a problem of determining when to send an update.

Dynamic Source Routing Protocol (DSR)

DSR is a source routed on-demand routing protocol. Every node maintains a cache of routes that is aware of and inserts or updates the cache when it learns of new ones. When a source node wants to send a packet to a destination, it first searches its route cache to determine if already contains a route to the destination. If an unexpired route exists, then that route is used. Else, if the route is unavailable or expired, the node invokes the route discovery procedure by broadcasting a route request (RREQ) packet. Each intermediate node checks the packet to see whether it knows a route to the destination. If it does not, it appends its address to the route record of the packet and forwards the RREQ packet to its neighbours. To limit the propagation of RREQ packets, a node only processes RREQ packets it has not already received before. A route reply is generated when the destination itself or an intermediate node with route information to the destination receives the RREQ packet. At the source, the route to the destination is read in reverse from the list of appended addresses in this reply packet.

Ad-hoc On-demand Distance Vector Routing Protocol (AODV)

AODV attempts to improve the table-driven DSDV protocol through minimizing the number of request broadcasts by creating routes only when required as opposed to DSDV that maintains the list of all routes. It uses a combination of both DSDV and DSR protocols. It uses the route-discovery and maintenance procedures from DSR but uses hop-by-hop routing, sequence numbers and beacons of DSDV. A hop-by-hop state of the route is stored in each node that is involved in the route as compared with the whole route being stored at the source in DSR.

Temporally Ordered Routing Algorithm (TORA)

TORA uses a "link reversal" algorithm to provide an on-demand but highly adaptive, efficient and scalable method of discovering and maintaining multiple routes to a destination. The main feature of TORA is that the control messages used are localised to a small set of nodes around the occurrence of a topological change (Park 1997). The protocol has three functions : route creation, route maintenance and route erasure.

Route creation is done using QRY and UPD packets. The algorithm starts with the 'height' (propagation ordering parameter of a node) of the destination set at 0 and the others set at NULL. The source sends the QRY packet with the destination node ID. Any node with a non-NULL height responds with a UPD packet with its height in it. Nodes receiving this UPD packet set its own height to one more above the value in the packet. This creates an acyclic route from the source to the destination.

If a link is broken, route maintenance is needed to re-establish the route to the destination. When the last downstream link of a node fails, it generates a new reference level and propagates this new information to the neighbouring nodes. The links then have their heights reversed to reflect the change thus reversing the flow direction of the links to a node. In route erasure, TORA floods a broadcast CLR packet throughout the network to erase invalid routes.

4.3.4 Function Calls and Module/Layer Relationship in NS

When running wireless simulations, there are different paths between the different layers that the code can follow in NS as depicted in section 4.3.2. To transmit or receive a packet, numerous function calls are made at different places, layers and times and each has its own task it must perform. An extremely simplified sample trace of a data packet being sent involving the MAC and PHY layers only is depicted below as an example.

1. When a data packet is received by the *MAC::recv( )* function in the MAC layer, it is checked to determine the direction of the packet. If the direction is DOWN, the packet originated from an upper layer and is meant to be sent out. The packet is then passed to the *MAC::send( )* function.
2. The *MAC::sendDATA( )* and *MAC::sendRTS( )* functions are called which builds the MAC header (for the data packet) and a corresponding RTS packet. The data packet with the MAC header is stored as *pktTx_* whilst the RTS packet is stored as *pktRTS_*.
3. Checks are made on the medium to see if idle and on the defer and backoff timers to ensure no other packet is being transmitted. If suitable, the defer timer is set (DIFS + slot time).
4. When the defer timer expires, the *MAC::check_pktRTS( )* function checks if any RTS packet is waiting to be sent. If yes, then the *TRANSMIT* function is called and the RTS packet is sent.
5. The packet is passed to the PHY layer using *downtarget_*. The *send_timer( )* and interface timer, *mhIF_* is started and the *tx_active_* flag is then set indicating the channel is busy.

6. When a CTS packet is received, the *MAC::recv( )* function is called again. If the packet is successfully captured (after the time period set by *recv_timer( )* elapses), the *MAC::recv_CTS( )* is called indicating that the destination node is ready to receive data.
7. Transmission then continues at the MAC layer as it prepares to send the data packet *pktTx_*. Similar to sending the RTS packet, checks for idle medium and idle defer and backoff timers are made. If suitable, the defer time is set for the data packet.
8. Once the defer timer expires, the *TRANSMIT* function is called once again to pass the data packet, *pktTx_* to the PHY layer to be transmitted.
9. Upon receiving an ACK for the data packet sent, the same process as above repeats but this time the *MAC::recvACK( )* is called to indicate successful transmission of the packet thus completing a round of transmission.

Note that many functions and tasks performed are not depicted in the example such as timer calls, queue checks, variable settings and such. It is also assumed that a *perfect* uninterrupted delivery of the packet occurs with no congestion, collision or errors experienced, all situations of which are simulated as additional function calls within NS. In addition, the events are depicted as a continuous sequence when in actual fact there are many delays between the time the RTS is sent, the CTS is received and when the data is sent and acknowledged of which other events can occur in between, all of which are controlled by timers and schedulers. Anyhow, it is clear that implementing a new function into NS would involve many parts of code at different places to ensure it works properly and yet does not adversely affect the other functions in NS.