CHAPTER 5

SYSTEM IMPLEMENTATION

This chapter details a proposal to solve the performance issues in WLANs using fuzzy logic. This is followed by the coding work done and steps taken to implement the proposed changes to the wireless model in NS. Since the code is too long to include in its entirety here, small sections are shown instead to highlight particular points of interest in the code whenever required. However, *diff* files of the implemented code are included in the appendix section for additional reference.

5.1 Prior Research into Improving WLAN Performance

The amount of research done in the wireless field is ever increasing due to its rising popularity but alas, the majority of them only entail typical evaluations and/or comparisons of the various wireless network protocols and standards. This includes the usual measurement of performance for the various types of network architectures, routing types and traffic types over a WLAN. Nevertheless, there has been a number of research works done that involve work into *improving* the performance and not just evaluating it *per se*.

Zygmunt J. Haas and Marc R. Pearlman attempts to improve the performance of WLANs by improving the quality of routes between nodes in an ad hoc network.

Their work involves the routing layer and implements a new routing protocol called the *Zone Routing Protocol* (ZRP) which is a hybrid between proactive and reactive ad hoc routing protocols (Haas 1997, Haas & Pearlman 1999). ZRP supports proactive discovery and maintenance of routes within a specified limited routing zone through its *Intrazone Routing Protocol* (IARP). The routing zone is determined as the number of hops from a particular node and is usually around two hops. This decreases the amount of periodic updates required to the nodes within the routing zone only and not depend on the size of the entire network which might be large.

While IARP maintains routes within the routing zone, the *Interzone Routing Protocol* (IERP) has the task of finding routes to nodes exceeding the routing zone radius. Instead of invoking the usual flood-search query and response protocols to locate routes, the IERP uses bordercasting which only involves the perimeter nodes of a routing zone. These perimeter nodes use IARP to check if the destination is within its routing zone. If so, a reply is sent back to the source indicating the route. If not, the peripheral node executes the same IERP procedure and bordercasts to *its* peripheral nodes. Any repeated queries are considered redundant and dropped using *Early Termination* even though it was bordercasted by a different nodes because each query is numbered and thus prevents repeated queries and helps reduce wasted transmissions and bandwidth which could have been used for other purposes.

Another research project using the routing layer is the *Explicit Link Failure Notification* (ELFN) by Gavin Holland and Nitin Vaidya which relies on information from the DSR ad hoc routing protocol (Holland & Vaidya 2002). The original DSR "route failure" message is modified to carry the ELFN message which is similar to a "unreachable destination" message in ICMP. Once the TCP source receives this message from a node, it proceeds to disable all its retransmission timers and enters a standby mode to prevent any further attempts to transmit to the node involved. Periodically, a route request packet is sent to probe the network to see if a route to the node is re-established. If so, the node leaves standby mode, restores retransmission timers and resumes transmission

The research of Ajay Singh and Sridhar Iyer involves the TCP layer instead with their implementation called ATCP (or Adaptation of TCP) which serves its functions between the TCP and IP layers (Singh & Iyer 2002). ATCP attempts to alleviate the degrading effect of mobility on TCP performance by using feedback information provided by the network layer, specifically the *connection* and *disconnection* messages, and retransmission timeouts (RTO) to avoid the congestion control mechanisms of TCP. The ATCP mechanism is complex and is dependent on whether the node is mobile or static and handles the feedback information differently.

From a mobile node, if a *disconnection* event is received, any outgoing packet queues and the retransmission timer (RTX) are stopped. If there are no transmissions but the node is awaiting an ACK, then the RTX is unchanged and left for a RTO event. Upon receiving a *connection* event, the RTX is reset and any outstanding transmissions are continued. If the node was awaiting an ACK and an RTO event has occurred, the previous packets are retransmitted else it continues waiting for the RTO event. Upon an RTO event, ATCP checks to see if a disconnection has occurred. If so, the slow start threshold (SST) is set to current the congestion window (CW) value and CW to one, else the lost packets are retransmitted without any changes to SST or

CW. If in a static node, upon receiving a *disconnection* event, the network connectivity status is updated. Upon a *connection* event, two ACKs are sent and the advertised window in each is set such that the RTX in a mobile node is first halted without shrinking the CW and then retransmission of unacknowledged packets is begun. These actions prevent any congestion control measures from invoking prematurely due to disconnection events.

Feng Wang and Yongguang Zhang's work also focus on the TCP layer to improve ad hoc network performance with their implementation *TCP-Detection of Out-of-order Response* (TCP-DOORS) (Wang & Zhang 2002). TCP-DOORS relies on the sequence number of a TCP packet or ACK packet to determine the current network congestion situation. Their argument is that with TCP, once a packet is delivered out of order, duplicate ACKs are sent to acknowledge received packets causing the sender to reduce its CW and SST by half and reset the RTO. The case becomes worse if the source timeouts while waiting for an ACK because this results in a slow start condition and thus deteriorating the TCP throughput. With TCP-DOORS, once an out-of-order condition is detected at either the source (through the ACK sequence number) or destination (TCP packet number), it can take one of two possible actions.

The first is just by halting the usual congestion control mechanisms temporarily so that the CW, SST and RTO values are kept at its current state. The second action sets the state at the sender at values prior to congestion was detected instead of fully resetting or halving them. The rationale behind either action is that the congestion that occurred has since resolved and therefore ACKs can be delivered but only in the wrong order. The congestion is assumed to be result of a route change and since

ACKs have resumed but only in the wrong order therefore TCP should not have to invoke slow start or linear window recovery.

Fabius Klemm and Zhenqiang Ye take an alternative approach to improving ad hoc WLAN performance by altering the MAC layer with their implementation of *Persistent MAC and Proactive Link Management* (Klemm et al. 2003). To improve TCP performance, Persistent MAC keeps track of neighbouring nodes with active routes based on the signal strengths of transmissions. This information informs a node about neighbouring nodes that are still within transmission range. Using this information, a node can then attempt to identify false link failures caused by congestion and try to re-establish connections with nodes using seven additional RTS-CTS handshakes. The reasons behind this action is that on a congested network, a node could be overhearing transmissions from other nodes and cannot reply to the RTS packets because it is being blocked from doing so. Extending the number of times to retry the RTS-CTS handshake would then increase the chances of the handshake succeeding.

In Proactive Link Management, the information of signal strengths between nodes is put to use once again. Once the signal strength drops below a certain level, it is assumed to be going out of range and therefore will lead to a route failure. It then stimulates the PHY layer to slightly increase the transmission power temporarily in attempt to keep the link to the neighbour alive. While the link is still active, a route request is then initiated to find a new route to the node before the current route fails in hopes of maintaining a continuous transmission path to the destination.

While it is pretty much impossible to fully control the levels of congestion in a wireless network, it is possible however to resolve most of the problems that arise from it. As shown above, prior research has attempts to improve the performance of WLANs by manipulating the transport, routing, MAC or PHY layers. For the most part, the strategies are typically designed for ad hoc networks because they lack the AP component found in infrastructured networks to help distinguish between losses caused by errors or congestion. The proposal for this thesis therefore takes a similar approach to improve ad hoc network performance by adapting existing MAC and PHY layers as well as combining them with several other strategies and concepts acquired from the research above.

The proposed method to improve WLAN transmission performance makes use of information obtained from the PHY layer – namely the signal strength of transmissions. The signal strengths of transmissions between nodes is particularly revealing since the signal strength is inversely proportional to the distance between the nodes. The information collected is stored at every node in a neighbour information table very much similar to the one implemented in the *Zone Routing Protocol*. Entries into the table however are only made for immediate neighbouring nodes and not for separate routes or next hop destinations therefore reducing the amount of entries required. In fact the maximum number of entries in the table would actually be the number of nodes in the network since that would be the highest amount of neighbouring nodes. Additionally, only successful transmissions are kept as a record in a neighbours table at each node. This would reduce memory and system demands at the node and simplify processing of the information table.

When a node fails during the RTS-CTS handshake, the MAC layer at the node could then figure out as to whether the failure was caused by an actual link failure (i.e. nodes moving out of range) or due to congestion using the recorded signal strengths. If the cause is due to link failure, then MAC abandons any further attempt to retry communicating with the node. The link layer could then be informed and depending on routing protocol used, the route is either removed from the routing table or a new route request is invoked. If the communicating nodes are deemed to be still within range, the failure is assumed to be caused by congestion. Since mobile nodes in an IEEE 802.11b network have a large interference range of around 480 metres radius (different manufacturers will have different ranges), it can overhear transmissions from quite a distance away and set its NAV to busy because of the exposed node problem. In this case, the MAC layer would then continue its attempts to retransmit the RTS in hopes that a handshake connection can be made to allow transmissions to proceed.

The default number of times to attempt the handshake is seven times but retrying an additional seven times (as in *Persistent MAC*) could be wasteful of both time and bandwidth, so a fuzzy logic controller is implemented into the model to evaluate the network situation and adaptively attempt the RTS retransmissions. When the handshake between two nodes fail, the information stored in the neighbour table is recalled and used to determine the likelihood that the nodes experienced congestion or actual link failure. Further details about the implementation of this proposal is described in the next sections.

5.2 Implementing Changes into NS

The work in this project can be roughly divided into 2 phases, namely, the incorporation of a IEEE 802.11b wireless network model and the fuzzy logic controller within that model. The following sections describe the modifications made to the main files that were involved. Other files in the source hierarchy were also modified but only slightly to the extent of adding function pre-headers or variable declarations so that they can be inherited and accessed in other files.

The NS v2.1b9a package by default only supports the original IEEE 802.11 wireless network model which is limited to bandwidths of 2Mbps and 1Mbps. Since this type of network is rarely used, work on implementing fuzzy logic in it is not as realistic and practical. Thus a new wireless simulation model is introduced into NS by incorporating the IEEE 802.11b protocol with support for transfer rates of up to 11Mbps. Most of the changes were made to the MAC and PHY modules in NS based on IEEE 802.11b published standards and documentations. Several of these modifications are based on a patch source offered by Siddhardtha Saha but with changes to incorporate additional information made available about inter-channel interference.

The newer IEEE 802.11a and 802.11g standards are avoided since it uses the ODFM technique for transmitting thus requiring a full rewrite of many portions of code since OFDM has yet to be implemented into this version of NS. On the other hand, the 802.11b standard is mostly based on the original 802.11 standard and thus the models can be readily inherited from those already present in the simulation package with only several changes required to certain sections in the code.

5.2.1 Implementing the IEEE 802.11b MAC Layer

Changes in the MAC layer have been made to the *mac-802_11.cc* and *mac-802_11.h* files in the NS source. First changes are to include the revisions in several time periods used in calculating arrivals and departures of packets to the MAC layer based on the published 1999 IEEE 802.11 protocol and 802.11b supplement. In the protocol, the SIFS and slot times are fixed per physical layer (IEEE 1999a) follows:

```
SIFS = RxRFDelay + RxPLPCDelay + MacProcessingDelay +
                                                 RxTxTurnaroundTime

SlotTime = CCATime + RxTxTurnaroundTime + AirPropagationTime +
                                                 MacProcessingDelay
```

From the SIFS and slot times, the PIFS and DIFS are derived through the following equations:

```
PIFS = SIFS + SlotTime

DIFS = SIFS + 2 × SlotTime
```

The final EIFS is derived from both DIFS and SIFS plus the time it takes to deliver one ACK control frame at 1Mbps following the equation:

```
EIFS = SIFS +(ACKsize × 8)+ Preamblelength + PLCPHeaderLength + DIFS
```

where *ACKsize* is given in bytes and "*(ACKsize x 8) bits* + *Preamblelength* + *PLCPHeaderLength* + *DIFS*" is measured in microseconds.

The IEEE has also predetermined the values of the certain constants (IEEE 1999a) although several of them have been left open to user implementation. The values used in the model are as follows

Table 5.1 – PHY characteristics as given in IEEE 802.11 standard

Characteristic	Time value
SlotTime	20μs
SIFSTime	10μs
CCATime	≤15μ5
RxTxTurnaroundTime	≤5μs
TxPLCPDelay	Any value as long as the requirements of *RxTxTurnaroundTime* is met
RxPLCPDelay	Any value as long as the requirements of *RxTxTurnaroundTime* is met
RxRFDelay	Any value as long as the requirements of *SIFSTime* and *CCATime* is met
AirPropagationTime	1μs
MACProcessingDelay	0μs (Not applicable)
Preamblelength	144μs
PLCPHeaderLength	48μs

To implement the changes, 5 new constants are defined in the MAC model header file as explained in brief below

DSSS_RxRFDelay – Time delay required for transmission to reach receiving end

DSSS_RxPLPCDelay – Time delay during which the physical layer convergence protocol operates to create a PPDU.

DSSS_MACProcessingDelay – A constant value used to take into account the delay caused by MAC processing.

DSSS_AirPropagationTime – The time a packet takes to propagate one way across the network.

MAC_AIR_PROPAGATION_CONST – Constant value set at 2 to represent a round trip of propagation time.

The original constant variable declared for *DSSS_SlotTime* is replaced with the appropriate values as in the given equation for slot time. *DSSS_SIFSTime* is removed entirely and the *sifs_* variable is instead initialised in the class constructor. The two sections of code involved are as follows with the changes highlighted in grey.

```
static PHY_MIB PMIB = {
   DSSS_CWMin, DSSS_CWMax, DSSS_CCATime + DSSS_RxTxTurnaroundTime +
   DSSS_AirPropagationTime + DSSS_MACProcessingDelay, DSSS_CCATime,
   DSSS_RxTxTurnaroundTime, DSSS_PreambleLength,
   DSSS_PLCPHeaderLength, DSSS_PLCPDataRate, DSSS_RxRFDelay,
   DSSS_RxPLPCDelay,DSSS_MACProcessingDelay, DSSS_AirPropagationTime
};

Mac802_11::Mac802_11(......){
// Other lines of code in constructor.........

// sifs_ = phymib_->SIFSTime; // This is original code
sifs_ =    phymib_->RxRFDelay + phymib_->RxPLPCDelay +
           phymib_->MACProcessingDelay + phymib_->RxTxTurnaroundTime;

// Rest of the code.........
}
```

The other major change to the MAC layer is in the function that keeps track of the RTS retransmissions. In the original code when the *RetransmitRTS* function is called, the *ssrc_* counter is incremented then checked. If the number exceeds the constant variable *ShortRetryLimit* which is by default set at 7 following the IEEE 802.11b protocol, the RTS packet is dropped and the contention window is reset. If the retry limit has not been reached, the RTS packet is prepared to be retransmitted by calculating the backoff time using the contention window value (*cw_*) after checking that the medium is idle.

In the modified code, the original version of this *RetransmitRTS* function is retained and can be accessed by setting the variable *fuzzyflg_* in the Tcl simulation script with the following statement (refer page 134, Appendix A):

```
        Mac/802_11 set fuzzylogic_ 0
```

This statement is evaluated when the class constructor is called to set the *fuzzyflg_* variable to 0 or 1 to disable or enable the fuzzy logic code respectively. Doing this allows simulations to be run either with or without the fuzzy logic code without requiring repeated editing, compilation and linking of the code.

When the *fuzzyflg_* is true, the fuzzy logic control code is executed. The code first determines if any additional retransmissions other than the basic seven have been performed by checking a variable called *retryflg_* which is always 0 unless the logic code has already been executed. If true, then the RTS packet is discarded the usual way and all *ssrc_*, *retryflg_* and *cw_* counters are reset. If false, then the code proceeds to evaluate the surrounding conditions to determine if any extra retransmissions are required. This step ensures that the code does not enter an infinite loop in retrying the RTS retransmissions.

To evaluate the neighbourhood situation, a function *evalretry* is called. This function is coded in the PHY model files and is called using the *netif_* downtarget from the MAC model to obtain the handle of the respective physical layer for each MAC. The *evalRetry* function returns an unsigned integer value stating the number of times to retry transmitting the RTS packet. Further details of this function is in the section below describing the PHY layer.

When the integer is returned, if it is more than zero then retransmission is needed else the RTS packet is discarded. To force retransmissions, the *ssrc_* variable is reset by subtracting the number of times to retry. This is followed by setting the *retryflg_* indicating that it had entered additional retransmission and resetting the contention window. If during the RTS retransmissions a CTS is received, then the *retryflg_* is reset to ensure it does not impede the next round to RTS-CTS exchange. The remainder of the class is not altered in any way.

5.2.2 Implementing IEEE 802.11b Channels

The original channel model in NS does not handle interference between the 14 available channels in the ISM band used for transmission. The semantics of the original NS channel model also differs slightly from the physical channels that are used in 802.11b. To overcome this difference, the *Channel* class in *channel.cc* and *channel.h* files have been subclassed into *Ch_80211* so as not to interfere with the original *WirelessChannel* subclass. The original implementation uses a fixed variable *delay_* to determine the time difference between when a node sends a packet and the time the destination node receives it. This period of time is used by the scheduler to manage when a node calls its *sendup* function to indicate it has received a packet.

This method is rather inaccurate because the delay should increase as the node moves apart and decrease as they approach each other instead of a fixed value.

In the modification, the static variable is replaced by a call to the function *get_pdelay*. This function uses an available function called *propdelay* from the class *MobileNode* to calculate the delay value. If the two nodes are on top of each other due to calculation, an additional constant value in *DBL_EPSILON* is used to 'move' the nodes fractionally apart so as to prevent the delay from being zero to prevent multiplication errors. The remainder of this subclass remains the same.

5.2.3 Implementing the IEEE 802.11b Physical Layer

The incorporate the new model, the original *WirelessPhy* class in the files *wireless-phy.{cc,h}* is subclassed into *Wireless_802_11_Phy.{cc,h}* to include the code for channel selection, channel interference and additional fuzzy logic coding. Other functions not overridden in this subclass remain the same and are inherited from the original *WirelessPhy* class.

To incorporate the channel selection feature, a new protected variable *channel_number_* is added and its value is bound in the class constructor to the default value of 10. Any number between 1 and 14 can be used, but channel number 10 is set as the default number in the *ns-default.tcl* file since vendors often set that value out of the factory. From this channel number, the frequency channel to be used is obtained for the *freq_* and *lambda_* variables used in later functions. The frequency channels have been predetermined in the IEEE protocol and are obtained by calling the function *getFreqFromChannelNumber* while passing it the channel number. Illegal channel numbers are rejected and the simulation fails following the coding style in NS once errors occur. The channel number can be changed by setting the *channel_number_* variable before a node is created with the following commands in the simulation script.

```
Phy/WirelessPhy/Wireless_802_11_Phy set channel_number_ 2
set newnode [$ns_ node]
```

Other variables inherited from *WirelessPhy* remain the same and are created when the parent class is referenced.

Whenever a packet is received from the channel, the *sendup* function in the PHY layer is called. From here the power of the incoming packet is calculated and compared against two thresholds; the carrier sense threshold (*CSThresh*) and receiving threshold (*RXThresh*). The carrier sense threshold signifies the range in which a node can overhear a transmission but cannot receive a packet successfully whereas the receiving threshold is the minimum power a packet must reach to be captured successfully. The original *sendup* function in *WirelessPhy* is overridden in the new subclass derived from it to add several new functions such as the code to store a neighbour node information table.

The first function, *calcChangedPr,* is added to determine inter-channel interference. Three variables are passed as parameters to this function – the incoming packet's power and the destination and source node's *lambda_* values. From here, the channel numbers are reverse-calculated and the amount of interference experienced is obtained. The interference amount is based on the results obtained from experiments conducted by Cirond Technologies Inc. in their unpublished white paper about channel interference in IEEE 802.11b networks (Cirond 2002). Although the actual calculation should take into account many factors to determine the actual amount of interference, the code written takes the easier approach by assuming the amount of power loss is in relation to the amount of interference experienced by node depending on the channel difference as in the table given below (Cirond 2002). Thus, the assumption is that a 27% of interference results in a loss of 27% of power. This is a huge assumption but one that will not impact the results of the end simulation at all because all the nodes will use the same channel when running in ad-hoc mode. The code is therefore implemented for simulation model completeness purposes only and facilitates future development.

Table 5.2 – Interference amounts between channels

Channel difference	Interference experienced	Actual power received (%)
1	0.7272	27.28
2	0.2714	72.86
3	0.0375	96.25
4	0.0054	99.46
5	0.0008	99.92
6	0.0002	99.98
7	0	100.0
8	0	100.0
9	0	100.0
10	0	100.0

To create a neighbour information table, a struct called *neighbors_* (American spelling for variables is maintained for consistency with other variables in the simulator) is added to the *wireless-phy.h* header file. It stores the values for a node's neighbours ID, the two most recent distances measured between the nodes and the transmission power of successfully received packets from the respective nodes.

When the *Wireless_802_11_Phy* constructor is called, a corresponding *neighbors_* struct object is created so that each node has its own neighbour table. The 'table' is in fact a linked list which is the method used in NS models to store information about multiple nodes such as in routing tables as opposed to using multi-dimensional arrays. Neighbour node entries in the table are updated accordingly whenever a packet is successfully received by a node from the channel (i.e. when the packet is above RxThresh and is successfully captured without errors and collisions). The pseudocode for the new *sendup* function is as follows

```
Wireless_802_11_Phy::Sendup(Incoming packet as P)
{
      Get current time and node's information
      Get handle to node's neighbour table
      If
      { No table exists, create new table set to null values }
      else
      { Calculate channel interference
      Calculate incoming packet P's transmission power as Pr
      If Pr does not meet CSThresh
            Discard packet and exit Sendup function
      If Pr does not meet RXThresh
            Set packet header to indicate receiving ERROR
      Record/Update neighbour node information in table}
      Capture packet and inform next layer
}
```

Only the time, transmission power and distance values are recorded for each node in the table. The transmission power is already provided in *Pr* and what remains is just to log this value each time it exceeds the *RxThresh* threshold, indicating that a transmission succeeded. To obtain the time value, a call to the global scheduler is made which returns the time the packet is received and is stored in the variable *currtime_* in the *neighbor_* struct. Any present value in *currtime_* is moved to *oldtime_* thus making the entries in the neighbour table always hold the last two transmissions.

The same procedure is made to record the distances between the nodes except a call to the function *CalcDistance* is made to determine the current distance between the nodes. *CalcDistance* uses the formula from the free-space propagation model to calculate this value and this function is actually based on the *Pr* function. Using this propagation model, the distance, *d*, between the nodes can be calculated as follows:

$$d = \sqrt{\frac{P_t G_t G_r \lambda^2}{(4\pi)^2 P_r L}}$$

where P_t is the default transmission power, P_r the received signal strength, G_t and G_r are the antenna gains of the transmitter and receiver; λ is the wavelength and L is the system loss experienced. The values of G_t, G_r, λ and L are constants in the NS simulator and are based on an Orinoco wireless card specifications (Lucent 2000). Changing to a different wireless network adapter's specifications only requires the appropriate changes to these constants to match the given specifications. Similar to recording the power values, the last two successful transmissions have their distance recorded into *currdistance_* and *olddistance_*. An example of the code for updating a neighbour node's information is shown below: The initial *for( )* loop is responsible for locating the correct entry in the link list to updated.

```
Time now=Scheduler::instance().clock();
for (ns = neighbors_ ; ns != NULL ; ns = ns->next_) {
     if (ns->nnode_ == p->txinfo_.getNode()->nodeid()) {
     ns->oldtime_ = ns->currtime_;     // Backup previous time
     ns->currtime_ = now;        // Replace current time
     ns->oldrcvdpwr_ = ns->currrcvdpwr_;
     ns->currrcvdpwr_ = Pr;      // Store rcvd pkt pr
     ns->olddistance_ = ns->currdistance_; // Previous distance
     ns->currdistance_= propagation_->CalcDistance(&p->txinfo_,
               &s, this, Pr); // Insert new distance
     }
}
```

If the *for* loop fails to locate a matching entry, then a new entry is added to the table. Similarly, when the table is newly created and no information exists, an entry is made into the table instead of updating existing entries. The information stored in the neighbour table is used in the fuzzy logic code as explained in the following section.

5.3 Incorporating Fuzzy Logic Control

The fuzzy logic code is separated into the MAC and PHY layer modules in NS to simplify and reduce function calls and variable passing to a minimum. This is also mainly because certain variables cannot be accessed from other classes due to their protected/private declaration in the original code – changing the type of variable to public affects other parts of code also using the variable, so this is avoided. The

Sugeno method of fuzzy logic inferencing is used since it is fast and avoids complex calculations and integrations thus making it uniquely suitable to the task because the MAC layer is busy enough with the current work it is required to do.

In the implemented fuzzy logic control, when a node N_A communicating with neighbour N_B exceeds the RTS retransmit limit while trying to establish the RTS-CTS handshake operation, it then proceeds to call the *evalRetry* function. The fuzzy logic code is implemented within this function and is passed a reference of the current packet to be sent as its parameter. This packet is used to obtain the destination node's ID and its corresponding entry in the neighbour table (of the node sending this packet) by traversing the link list to match the node ID found in the packet header.

If no node entry is found, the attempt to retry retransmitting RTS is abandoned since it is obvious that the node never had a route to the destination. The termination process is similar to the original. If a matching entry is found, the entry's table values are checked to see whether they have expired. The expiry time is set at 8 seconds which might seem rather short but it is chosen because the maximum speed of node movement used during the simulations is at 20m/s. If a <u>mobile</u> node moves away from a <u>static</u> node at 20m/s, within 8 seconds it would have moved 160 metres apart which is the maximum transmission range of the nodes. Therefore, after 8 seconds a node would theoretically be at the boundaries of the transmission range anyway. This assumption is valid only if the pair of nodes' starting coordinates are the same (one node on top of the other perhaps), the nodes move in a linear fashion and only <u>one</u> node moves in the pair. If both nodes are already far apart and/or both nodes move apart simultaneously and/or even if the nodes move at right angles to each other, then the distance covered would certainly be greater within that 8 second period (based on theory of relative velocity). If the entry has not yet expired, then the current distance between the source and destination nodes is estimated using the following equation.

```
Estimated distance = Last recorded distance + node speed x (time
                          elapsed between now and last entry time)
```

Letting the two latest entries in node N_A's neighbour table for node N_B be *d1* at *t1* and *d2* at *t2* for the distance and time where $t1 < t2 < t$ and *t* is the current time, as a result:

Estimated distance, d_{EST} = d2 + [(d2 - d1) / (t2 - t1) x (t - t2)]

This estimated distance value is the first input to the fuzzy logic inferencing process. The second input is the speed at which the nodes are moving relative to each other. Using the difference in distance and recorded times, the velocity is determined to be either getting closer or moving farther apart.

With these two inputs, the fuzzy inferencing process can then begin. The inferencing process uses 4 fuzzy relations to fuzzify the inputs. The relations can be approximated into sentences as follows:

If *nodes* are *near* AND nodes are *separating* THEN retry times is *medium*

If *nodes* are *near* AND nodes are *approaching* THEN retry times is *medium-high*

If *nodes* are *average* AND nodes are *separating* THEN retry times is *low*

If *nodes* are *average* AND nodes are *approaching* THEN retry times is *high*

A *high* number of retries is set at seven times and *low* at a single retry. *Medium* and *medium-high* is set at three and five additional retries respectively. Therefore, the number of additional retries is limited to seven times therefore making a total maximum of 14 retries each time (inclusive of the initial default seven) before the handshake attempt is actually terminated.

To represent the fuzzy sets, arrays are used where the array elements represented points of the fuzzy relation graph. The distance array has seven elements of which the first one identifies whether it is a trapezoid or triangular relation to allow for easier changing of fuzzy relation types. The following four elements are the points in the graph from left to right, while the last two tells if the two ends allow for infinite values beyond the graph limits. The second array for velocity only uses six elements only because all of them use the trapezoidal shape and does not need the first identifying element unlike the distance array. The array declarations used are shown below alongside their graphical representations in the corresponding figures.

```
Float myrange[2][7] = {
                    {1.0,  0.0,  5.0, 55.0, 60.0,0.0,0.0},
                    {1.0, 55.0, 60.0, 65.0,160.0,0.0,0.0},
          };
```

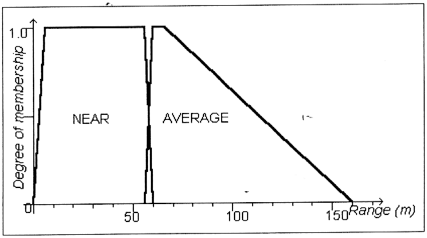

Figure 5.1 – Fuzzy membership graph for range

```
double myaccel[2][6] = {
                {0.0,0.0,0.1,10.0,0.0,0.0},
                {-10.0,-1.0,0.0,0.0,0.0,0.0}
        };
```

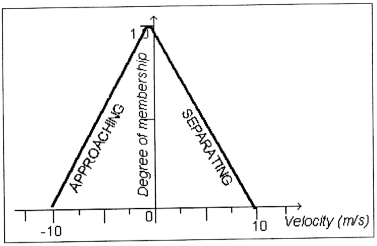

Figure 5.2 – Fuzzy membership graph for velocity

Since the fuzzy relations above use the AND operator on the consequents, the min operation is used on the results of the input fuzzification so the smaller value between the estimated distance and velocity is chosen during the inferencing stage. The weights used in the Sugeno defuzzification are relative to the output values required (i.e. medium, high, low) and are multiplied with the results of the inferencing stage. The final result is cast to an integer and returned.

```
    // Inferencing
    for (i=0;i<6;i++)
    {
        if (preout2[i] < preout1[i]) // AND (if OR then use '>' )
            foutputHeight[i]=preout2[i];
        else
            foutputHeight[i]=preout1[i];
    }
```

```
// Defuzzification
for (i=0;i<6;i++)
{
        totalY+=myretry[i]*foutputHeight[i];
        totalH+=foutputHeight[i];
}

if (totalH>0)
        finalOutput=totalY/totalH; // Avoid div by zero
else
        finalOutput=0;

return (int)(finalOutput);
```

Regular testing and verification of the modified code is important to make sure that the code runs correctly and optimally and also to allow for trial runs to fine-tune the weights involved in the defuzzification. Since running the fuzzy logic code in the simulator would require a long period of time, initially a standalone version of the logic code was created in C code to accept two inputs entered during run time to emulate the simulator passing two inputs to the code. Everything in the standalone version remains the same except values for the power of received packets was changed to integer values to simplify entering input values. Further testing of the code was incorporated into the process of fine-tuning the weights and fuzzy relations of the fuzzy logic code during test runs of the simulation.