Chapter 5
Simulation Results and Analysis
Chapter 5
Simulation Results and Analysis

The protocols developed as part of this thesis work were simulated using the OMNeT++ simulator as it was costly to build a testbed of suitable size to validate these protocols in the envisaged application scenario. Network simulators provide researchers with a virtual and reproducible environment where they can design, configure, run and analyze different kinds of networks without getting their hands dirty with real-life hardware. Hence, researchers can concentrate on their study and test their developments easily. The three steps in the simulation of a network application basically include the development of the protocol, the creation of a simulation scenario (e.g. designing a network topology and traffic scenario), and the selection of statistics to be collected. The last step is the visualization and analysis of simulation results which may be carried out during or after the simulation execution. The Mobility Framework of the OMNeT++ simulator was used in our work as it presented the required models for designing the network topology and traffic scenario in WSN.

The section 5.1 describes the Mobility Framework in OMNeT++ simulator with regard to our application. Section 5.2 describes the simulation model used in MLMAC protocol followed by the results and analysis of the protocol in section 5.3. Section 5.4 deals with the SOMAC protocol simulation model and section 5.5 presents the results and analysis of the protocol. Section 5.6 discusses the simulation model of SOR Protocol followed by results and analysis of the protocol in section 5.7. Section 5.8 deals with the DMPR protocol model and the results and analysis of the protocol is discussed in section 5.9. Section 5.10 concludes the chapter with the inferences drawn from the modeling simulation of the protocols in the simulator.
5.1 Introduction to Mobility Framework in OMNeT++ Simulator

This Mobility Framework (MF) [91] is intended to support wireless and mobile simulations within OMNeT++. The core framework implements the support for node mobility, dynamic connection management and a wireless channel model which makes it ideally suited to our simulation of protocols in WSN. Additionally the core framework provides basic modules that can be derived in order to implement own modules. With this concept a programmer can easily develop own protocol implementations for the MF without having to deal with the necessary interface and interoperability issues [93]. The framework can be used for simulating:

- fixed wireless networks
- mobile wireless networks
- distributed (ad-hoc) and centralized networks
- sensor networks
- multichannel wireless networks
- many other simulations that need mobility support and a wireless interface

There is a rich library of protocols to enable easy plug-and-play simulations of various kinds of widely used protocols. The MF structure is described in the section 5.1.1. The modules of MF used in our work are discussed in section 5.1.2.

5.1.1 Structure of the Mobility Framework

In order to have clearly defined interfaces that are easy to understand and extend if necessary, the Basic* module has been provided for each layer which in turn is derived from the global BasicModule of the OMNeT++. The concept of a Basic* module is to have a base class which takes care of the necessary work that has to be done but is not of specific importance for the real functionality. The general derivation structure of a Mobility Framework module is shown in Fig.5.1. The Basic* modules also provide a very basic functionality, eg., the BasicNetwLayer is capable of forwarding messages to and from upper and lower layers but has no routing functionality at all yet. The idea is to have the possibility to easily extend or adapt modules of different layers to the specific requirements of the simulation.
To serve this purpose two kinds of functions are defined: handle\*Msg() functions and convenience functions.

**handle\*Msg() Functions:** The handle\*Msg() functions contain the actual protocol functionality. They are called each time a corresponding message arrives and contain all necessary processing and forwarding information for messages where required. handleSelfMsg is the easiest way to implement timers in OMNeT++ which are self messages. handleSelfMsg() thus is the place to handle all timer related things and to initiate actions upon timeouts. handleUpperMsg is called every time a message has arrived from an upper layer. The message already has the corresponding layer n format (i.e. it is already encapsulated). After processing the message, it can be forwarded to the lower layers with the sendDown() function, if necessary. handleLowerMsg is for messages from lower layers and if needed, they can be forwarded to layer above it. This is done by using the sendUp() function which also takes care of decapsulation.

**convenience Functions:** The convenience functions are defined to facilitate common interfaces and to hide inevitable interface management from the user. Here three different functions are provided:

![Diagram](image-url)

**Figure 5.1:** The general module derivation structure
encapsMsg is called right after a message has arrived from the upper layers. It is responsible for encapsulation of the layer n+1 message into a layer n message. This implies to provide all necessary header information and if applicable also the conversion of layer n+1 header information into layer n information. After this the message is passed to the handleUpperMsg() function. sendUp is the function to be called if a message should be forwarded to upper layers and is usually called within handleLowerMsg(). It decapsulates the message before sending it to the layer n+1 module. Sending messages to layer n-1 is done with the sendDown() function. Sometimes it may be necessary to also to provide or process additional meta information. In the case of the network layer for example it may be necessary to provide a next hop address. The network layer destination address usually contains no information about the next-hop MAC address a message has to be forwarded to on its way to the destination so it has to be translated. sendDelayedUp is the function used to delay the point of time the message is sent to the upper layer. The time the messages should be delayed can be given as a parameter. sendDelayedDown is the same as sendDelayedUp, but for sending delayed messages to the lower layer.

These six functions are provided with slide differences in the Basic* module for each layer of the MF. Usually the three handle*Msg() and the initialize(int) and finish() functions are the only functions a programmer has to re-implement to create his/her own module and the functionality of it. The convenience functions should be used to serve their tasks so that newly implemented modules remain compatible with any other module implemented for the MF. The convenience functions cannot be changed in derived modules.

5.1.2 MF Modules used in the Simulation of Protocols
In the simulation of WSN, the simulator has to process mobility information and handle the connections between the nodes efficiently. A centralized approach is used for connection management since knowledge of the positions of all nodes is needed for this. Mobility is handled in a distributed manner since decisions how to move neither affect other nodes nor do they require global knowledge, as long as
the current position is known. Fig. 5.2 shows the communication and control relationships between the different modules. The core of the mobility architecture is the **Channel Control** module, which dynamically sets up and tears down connections between nodes, depending on distance and the physical characteristics of the nodes. Movement is controlled by independent **Mobility Control** submodules in each node.

The **Channel Control** module takes care of establishing communication channels between nodes that are within communication distance and tearing down the connections when they move out of range. MF has the concept of links between nodes, as opposed to direct message passing, since visible communication paths are an important source of information in early development stages.

With object-oriented development it seems natural to make each node responsible for its movement. This helps to keep mobility state local and allows independent mobility patterns for each node. A **Mobility Controller** submodule is defined for each host to handle mobility in the nodes. The **Mobility Controller** recalculates its position regularly and updates the graphical representation of its **Host** super module. It also communicates the current location information to the **Channel Control**. This communication is simply a function call, to avoid the overhead of another communication gate between the **Host** and the **Channel Control** and a message creation and destruction for each location update.

![Figure 5.2: Architecture for mobility support and dynamic connection management](image)
5.1.2.1 The Structure of a Mobile Host

The overall structure of the host module has two basic sources: the ISO/OSI architecture shown in Fig.5.3 and real-life hosts shown in Fig.5.4. The application layer, transport layer, and network layer are represented by compound modules. Below the network layer are one or more network interfaces. Each of these modules is representing a particular network interface of a host. A radio access point, for instance, will have at least two interfaces: one for the radio access and one for the fixed network connection.

Some of these interface modules, especially the radio interface, have to be accompanied by other modules that coordinate their communication. To achieve a "plug and play" use of these modules and allow others to use their own modules it is necessary to accurately define the interface for each module.

5.1.2.2 Blackboard Access

To enable anonymous and group communication, the concept of a Blackboard is introduced. This feature is to be used by submodules within a host to exchange information across layers, not unlike interprocess communication in a real computing device. There is exactly one Blackboard module per host. Modules that wish to use the Blackboard have to be derived from the BlackboardAccess class which itself inherits from cSimpleModule.

Communication takes place using the publish-subscribe paradigm. Modules that provide certain data (like a physical layer link sensing for carrier sense MAC protocols) can publish these on the Blackboard, using the publish() method family defined in the BlackboardAccess class. Internally, publications are mapped onto a sendDirect() call, so subscribers can actually find out who the publisher was.

Modules that are interested in certain types of information can subscribe for a certain topic using the subscribe() method of BlackboardAccess and will receive a notification message when matching data is published. The published data itself is
not sent along with the notification message, but is stored in the Blackboard module and can be accessed via a lookup() method. This allows both asynchronous lookups and multiple access to the data.

This feature has proven very effective especially when integrating modules from different sources, and leads to an efficient coding style.

![Figure 5.3: Structure of a Mobile Host](image)

5.1.2.3 The Nic Concept

A nic is a network interface card that includes physical layer functions like transmitting, receiving, modulation as well as medium access mechanisms. The nic module in the MF therefore is divided into a physical layer modules snrEval and decider and a MAC layer (macLayer). The snrEval module can be used to compute some SNR or SIR information for a received message whereas the decider module can process this information to decide whether a message got lost, has bit errors or
is correctly received. Therefore the decider only handles received messages and not messages that should be sent. The corresponding compound module with its simple modules is shown in Fig. 5.5.

The reason for putting the physical and the MAC components into one compound module is easily explained. For most lower layer protocols the MAC and the physical layer have to be coordinated, so for one protocol (e.g. IEEE 802.11) there will be a corresponding snrEval module as well as a corresponding decider module as well as a corresponding mac module. So to run a certain routing protocol over a certain PHY/MAC protocol it is only needed to choose the corresponding nic module when building the host. In the following the physical layer modules are explained in more detail.

![Figure 5.4: The structure of a practical node](image)

Figure 5.4: The structure of a practical node
5.1.2.3.1 snrEval
The structure of the snrEval module is a little different from those of the other modules. The handleLowerMsg() function is split into two functions in order to simulate the transmission delay. They are handleLowerMsgStart and handleLowerMsgEnd. The handleLowerMsgStart function is called in the moment the reception begins, i.e. in the moment the first bit arrives. Everything that is necessary to be done at the start of a reception can be done here, e.g. create and initialize an SNR-list to store SNR values, put the frame into a receive-buffer etc. The handleLowerMsgEnd function is called when the transmission of a message is over. Whatever is necessary before the message is handed on to the decider, e.g. take the message out of the receive-buffer, call the sendUp function etc., can be done in this function.

Figure 5.5: The structure of a nic module
5.1.2.3.2 The Decider Submodule

For investigation of many protocols, the pure bit error rate in the physical medium is sufficient. With others, like IEEE 802.11b, where headers are transmitted at a different rate than the packet bodies, a closer look at the transmission characteristics is necessary. To support different levels of abstraction in the physical layer, the concept of a decider module is introduced. While the physical layer module is responsible for recording the signal and noise power at different points in time (at each transmitted signal), the decider module uses that information along with the applied modulation scheme to calculate the bit error rate or to mark erroneous bits if needed. This decoupling of bit error simulation from transmission also allows to reuse the physical layer module with different modulation schemes.

The Decider module only processes messages coming from the channel (i.e. from lower layers). Messages coming from upper layers bypass the Decider module and are directly handed to the SnrEval module. Decisions about bit error or lost messages only have to be made about messages coming from the channel. Consequently there is no need to process messages coming from upper layers in the Decider module.

The Decider module takes the SnrList created by the SnrEval module and translates the SNR values to bit errors. The simplest possible implementation would be to compare the SNR values against a SNR threshold. If at least one of the SNR values contained in the SnrList exceeds the SNR threshold the message is dropped due to bit errors. The Decider would also be the place to implement error detection and correction codes.

5.2 Simulation Model for the MLMAC Protocol

The WSN node is modeled as in Fig.5.6a. The application layer was selected as sensorApplLayer of MF, generating data at a constant rate suited to voice streaming application. The net module was selected as FixedRoute module where the nodes are predecided for forwarding data to the sink externally before the
application starts, as shown in Fig. 5.6b. The MLMAC module in the node is shown in Fig. 5.6c.

Figure 5.6: Node Structure in MLMAC Simulation
(a) Main Modules in MLMAC simulation (b) FixedRoute module in net module

Figure 5.6c: MLMAC module in the node
The MLMAC protocol has been designed over the LMAC protocol which is also simulated in the MF. The message interactions of MLMAC to external layers on top and bottom layers are shown in Fig. 5.7. FixedRoute protocol is the upper layer protocol doing the routing functions in the nodes in the simulation, which sends packets to the MLMAC layer through its own sendDown(msg) function. This message is handled by the function handleUpperLayerMsg(msg) in the MLMAC layer of the node. The packets received from the PHY layer and processed in the MLMAC layer will be forwarded to the network layer if necessary through the sendUp(msg) function. Similarly, the packets to be send to the next hop node will be passed on to the PHY layer through the function sendDown(msg). The packets received from other nodes by the PHY layer would be processed in the function handleLowerLayerMsg(msg). The PHY layer of the MLMAC protocol is set parameters that are as per the IEEE 802.15.4 radios. The radio models and PHY models are shown in Fig. 5.8a, 5.8b and 5.8c. The internal working and status keeping of the protocol is with the help of internal messages described below:

MLMAC_WAKEUP: This message is generated for every slot time. During the handling of this message, the node finds the slot number and beginning of new frames as per the designed frame size. If the slot number is owned by the node, it transmits control packet after sensing the channel after a small random time which is set by a timer generating the message MLMAC_CHECK_CHANNEL. Else the node will listen for the control packet from other nodes. It also checks the time between consecutive packet arrival so that streaming activity can be detected in the node and change the schedule accordingly. If the node is in the streaming state, from the tierNum and the slot number, the current state of the radio is decided as per the MLMAC protocol.

MLMAC_SEND_DATA: If not in streaming state, the node is in its own slot and the control packet has already been sent, the next packet from the queue is prepared with the MAC header and sent. The nodes in the streaming state go ahead with transmission of data packet taken from the queue irrespective of the fact that the slot belongs to it.
MLMAC_CHECK_CHANNEL: The node checks the channel for transmitting the control packet. If the channel is free and the node is not in streaming state or not in the hearing range of streaming transmissions, the node will send its control packet. If the node is in streaming state, irrespective of the channel busy, the node will go ahead with the transmission of control packet. If a node finds the channel to be busy, there is a collision in the slot and the node will try to find an alternate slot as its transmission slot. If it cannot find a free slot, it will give up transmission for some time and try again.

MLMAC_TIMEOUT: This message is activated to receive the control packet in the beginning of a time slot. If some radio activity is detected in the beginning of
the slot, the receive time is extended by the timer for timeout period and wait for the control packet reception.

Figure 5.8a: SnrDecider Module of the MLMAC Radio Module

Figure 5.8b: SnrEval Module of the MLMAC Radio Module
The functions implemented in MLMAC protocol are as described below:

handleUpperMsg(cMessage *msg): The message is passed on to the MLMAC layer from the network layer and it has to be queued if another message is waiting to be send or if the node is already trying to send another message. If the packets are coming at streaming rate decided by the application, the node enters the streaming state and indicates this in all its transmissions. If the queue is full, the packet is dropped.

handleLowerMsg(cMessage *msg): If control packet is received, calculate the tierNum of the node if the packet has come from a node lower to the present tierNum. Also the node occupancy in the neighborhood is also mapped from the control packet information. If the node is addressed by the control packet, stay awake to receive the data packet, else switch off the radio and sleep during the rest of the time period. In streaming state, even if the control packet is corrupted, the node has to stay awake to receive the data packet. The code segments indicating the main functions performed in them are shown in Fig.5.9.
void MLMACLayer::initialize(int stage)
{
    timeout = new cMessage("timeout");  // Timer for random timeout
    timeout->setKind(MLMAC_TIMEOUT);  // before sending

    sendData = new cMessage("sendData");  // For start sending data
    sendData->setKind(MLMAC_SEND_DATA);  // after sending control pkt

    wakeup = new cMessage("wakeup");  // Starting of a new slot
    wakeup->setKind(MLMAC_WAKEUP);

    // End of setup phase. Change slot duration to normal and start sending data packets.
    // The slots of the nodes should be stable now.
    initChecker = new cMessage("setup phase");
    initChecker->setKind(MLMAC_SETUP_PHASE_END);

    // Check the channel in own slot. If busy, change the slot. If not, send a control packet.
    checkChannel = new cMessage("checkchannel");
    checkChannel->setKind(MLMAC_CHECK_CHANNEL);

    scheduleAt(slotDuration*5*numSlots, initChecker);
    scheduleAt(slotDuration, wakeup);
    SETUP_PHASE = true;
}

void MLMACLayer::handleUpperMsg(cMessage *msg)
{
    // Received a new message from upper layer for transmission
    if (macQueue.size() <= queueLength)
    {
        macQueue.push_back(mac);
        if (!inStreaming)
        {
            if (IntArrTime != 0)  // Check if 4 messages have been
CntPktStream++: // received at streaming rate

if (CntPktStream >= 4)
{
    mStreaming = true; // Mark the node to be in
    pktType = 1; // streaming state.
}
else

    IntArrTime = 17;

}

else {
    // queue is full, message has to be deleted
    mac->setName("MAC ERROR");
    mac->setKind(NicControlType::PACKET_DROPPED);
    sendControlUp(mac);
}

} MLMACLayer::handleSelfMsg(cMessage *msg)
{

    if(msg->getKind() == MLMAC_SEND_DATA)
    {
        // If not in streaming, node should be in its own slot and the control
        // packet should be already sent for sending its own data from queue.
        // Receiving neighbors should wait for the data now.
        // In streaming, send data from mac queue if mac is in transmit substate.
        
    }

    if(msg->getKind() == MLMAC_CHECK_CHANNEL)
    {
        // If not in streaming state, if the channel is clear, get ready for
        // sending the control packet. Else, some other node is using the slot
        // Hence find a new slot for the node.
        // In streaming state and its transmission slot, go ahead.
    }

}
if(msg->getKind() == MLMAC_WAKEUP)
{
    // If not in streaming and own slot, listen to channel after random timeout. Else receive in the slot.
    // If in streaming state, find mac substate as per equations 3.1 to 3.3. Switch radio accordingly.
}
}

void MLMACLayer::handleLowerMsg(cMessage *msg)
{
    if (mac->getKind() == MLMAC_CONTROL)
    {
        // If not in streaming and the node is addressed, receive the following data packet.
        // If the node is not in streaming, receive the data packet anyway.
    }
    else
    {
        // If the destination address is itself, send it up. Else, discard and keep the streaming on.
    }
}

Figure 5.9: The code segments in MLMAC protocol

5.3 Simulation Results and Analysis of MLMAC Protocol

The simulation parameters are specified in MF through in omnet.ini file. The main parameters used in this simulation is listed in Table 5.1. The simulation was run for 5000 seconds in a playground size of 1500m x 1500m deployed with 40 nodes. Since fixed routing was used in the network layer of the hosts, the nodes were deployed at specific points in the playground to ensure connectivity. The node deployment resulted in 6 tiers between source node (SN) and the BS. The radio was specified to have 2.445GHz carrier frequency, operating with maximum
power of 0.5 W and receiver sensitivity specified to be -110 dB. The application in one of the nodes (node 8) was to generated 2000 packets periodically with a period of 0.25 s. The application will start after 60 s so that nodes would initialize to provide data transfer in the network. The MLMAC layer was to operate on 0.04 s slots with 24 slots in the TDMA frame.

Table 5.1: Simulation parameters of MLMAC Protocol

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>sim-time-limit</td>
<td>5000 s</td>
</tr>
<tr>
<td>sim.playgroundSizeX</td>
<td>1500</td>
</tr>
<tr>
<td>sim.playgroundSizeY</td>
<td>1500</td>
</tr>
<tr>
<td>sim.numHosts</td>
<td>40</td>
</tr>
<tr>
<td>sim.channelcontrol.carrierFrequency</td>
<td>2445e+6</td>
</tr>
<tr>
<td>sim.channelcontrol.pMax</td>
<td>0.50</td>
</tr>
<tr>
<td>sim.channelcontrol.sat</td>
<td>-110</td>
</tr>
<tr>
<td>sim.MLMACHost[8].appl.nbPackets</td>
<td>2000</td>
</tr>
<tr>
<td>sim.MLMACHost[0..39].appl.nbPackets</td>
<td>0</td>
</tr>
<tr>
<td>sim.MLMACHost[*].appl.trafficType</td>
<td>&quot;periodic&quot;</td>
</tr>
<tr>
<td>sim.MLMACHost[*].appl.trafficParam</td>
<td>0.25</td>
</tr>
<tr>
<td>sim.MLMACHost[*].appl.initializationTime</td>
<td>60 s</td>
</tr>
<tr>
<td>sim.MLMACHost[*].nic.mac.slotDuration</td>
<td>0.04 s</td>
</tr>
<tr>
<td>sim.MLMACHost[*].nic.mac.controlDuration</td>
<td>0.02</td>
</tr>
<tr>
<td>sim.MLMACHost[*].nic.mac.numSlots</td>
<td>24</td>
</tr>
</tbody>
</table>

The delay performance of MLMAC protocol was analyzed by recording the time between the first and last data packet handled by the nodes in the streaming path. Fig. 5.10 shown the streaming transmission time in nodes in every tier in the streaming path. It can be seen that the 2000 packets from source, generated at a constant rate of 0.25 s between packets were transmitted within 499 seconds, the time is less than 500 s because the source queues up 5 packets to detect streaming activity. It is hence possible to capture the slots during streaming from neighboring nodes to increase the data rate to support streaming in this protocol. The time between first and last packets received in the streaming nodes in different tiers is plotted in Fig. 5.11. The figure shows that the packets are received within the total generation time of these packets in the source.
5.4 Simulation Model for the SOMAC Protocol

The streaming optimized MAC protocol used a new MAC module called RMLMAC in the host’s nic module. It also used a routing protocol called StreamingRoute in the net module of the host as shown in Fig.5.12. The RMLMAC layer in the nic module is shown in Fig.5.13. Rest of the modules in the host were kept the same as that of the hosts in the MLMAC simulation. The structure of the simulation code also remained except for the implementation of the functionalities of the new MAC protocol in the message handling modules. The changes in the implementation compared to MLMAC implementation are shown in Fig.5.14.
Figure 5.11: Total transmission streaming time in different tiers

Figure 5.12: Network protocol Module in the Host
void UCAMACLoyer::initialize(int stage)
{
// The subslots during streaming operation decided by self message.
subSlotStreaming = new cMessage("subSlotStreaming");
subSlotStreaming->setKind(MLMAC_STREAMING_SLOT);
}

void IJCAMACLayer::handleSelfMsg(cMessage *msg)
{
    if (msg->getKind() == MLMAC_STREAMING_SLOT)
    {
        // If not a new slot, use subslots to operate in streaming.
        // Check streaming status to continue in subslot cycle.
    }
}

Figure 5.14: Changes in implementation of SOMAC compared to MLMAC

A new self message MLMAC_STREAMING_SLOT and a timer subSlotStreaming to trigger that message were initialized in the SOMAC...
implementation. During the normal operation, the node would wake up in the beginning of the slots and function the same way as in MLMAC protocol. Once streaming activity is detected, the node activates the subSlotStreaming timer to generate the self message MLMAC_STREAMING_SLOT at reduced time intervals thereby dividing the normal slots to much smaller subslots to suit the voice streaming activity in the network and also to save power during the inactive period. The streaming slot and normal slot times are selected such that normal slot time is a multiple of 3 x streaming slot time. This is necessary for a node in the streaming state to send the control frame in the beginning of a normal slot.

A node after receiving the MLMAC_STREAMING_SLOT message would check whether it is a new normal slot and if not, the subslot streaming procedure is invoked to decide whether it is a TX, RX or ACK subslot. If the subslot is the beginning of a new normal slot, the subslot streaming procedure is bypassed and MLMAC streaming procedure is allowed in the node. By this method, this protocol works as an overlay on the MLMAC protocol to work with different slot timings for different operations in the network.

5.5 Simulation Results and Analysis of SOMAC Protocol

The simulation parameter used for the SOMAC protocol is shown in Table 5.2. 120 nodes were deployed randomly in a playground of size 100m x 5000m with time limit set for 3000 s. The radio was set to have a carrier frequency of 2.445GHz with maximum transmission power of 0.2W. The receiver sensitivity was set to -110dB. Node 8 was set as SN deployed at 1100 to 1200m from BS. Other nodes were randomly deployed within 0 to 1200m. Application layer was set to generated 10000 packets periodically with a period of 0.018s. The MAC slot was 0.06s and subslots were of 0.006s. The number of slots in the TDMA frame was 64.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>sim-time-limit</td>
<td>3000s</td>
</tr>
<tr>
<td>sim.playgroundSizeX</td>
<td>100</td>
</tr>
<tr>
<td>sim.playgroundSizeY</td>
<td>5000</td>
</tr>
</tbody>
</table>

Table 5.2: Simulation Parameters for SOMAC
The delay performance of the SOMAC protocol was analyzed based on the total transmission and reception times in the streaming nodes. The total transmission time in streaming path nodes in different tiers is shown in Fig.5.15. There is a delay in tuning to the new subslot structure for streaming and approximately 10 packets per tier and hence the number of packets received in the sink is 100 packets less than SN. Therefore, the total time of streaming data transmission in SN is more than in last hop node in tier 1. The difference in transmission times in nodes in different tiers is 1.51s which can be compensated by buffers.
The total reception time in streaming in nodes of different tiers is shown in Fig. 5.16. Here again since the number of packets received in the BS is 100 packets less than the SN, the total reception time in the BS is less than SN. However, this plot shows that the transmission is not delayed and the protocol is able to capture the slots of the neighbor to provide the streaming data rate in the network.

The throughput of the protocol is shown in Fig. 5.17 as the number of packets received in every tier for a total of 10000 packets transmitted from the BS. The BS receives total 9906 packets through 10 hops. The loss is due to the streaming initiation delay in different tiers. Therefore, to start the streaming, the SN should wait for an average of 10 packets per hop. This is acceptable in the application as there would be a setup time required for such applications.
Figure 5.16: Total transmission streaming time in different tiers

Figure 5.17: Total packets received by nodes in different tiers
5.6 Simulation Model for the SOR Protocol

The Streaming Optimized Routing protocol was simulated in the same MF with the ORMLMAC layer which was a modified version of the MLMAC protocol which is shown in Fig.5.18. The network layer protocol is shown in Fig.5.19 in the simulation host. The application is selected to produce data at voice streaming rate in one of the hosts. The StreamingRoute layer in the net module interacts with the MAC protocol through the handleLowerControl(cMessage *msg), and handleLowerMsg(cMessage *msg) functions. It interfaces to the application layer through the handleUpperControl(cMessage *msg), and handleUpperMsg(cMessage *msg) functions. The protocol is implemented with the various self messages generated inside the routing layer through timers. The data passing to the application and MAC layers are done using the sendUp(msg) and sendDown(msg) functions respectively.

Figure 5.18: MAC module in the SOR simulation host
The main code segments in the implementation of SOR protocol is shown in Fig. 5.20 which contains the main functions and comments in the code. The ROUTE_BROADCAST message is a self message to announce the existence of a route to BS from that node. Any node after establishing a route to BS will periodically broadcast a route for all other nodes above it. This route broadcast will stop when the node enters streaming state or when it hears streaming in its neighbourhood. StreamingTimer is to check whether data packets are received in the routing layer at the streaming rate. If the application layer publishes streaming state in the blackboard, the routing layer is subscribed to it and would enter the streaming state.

When the data packet is received from the MAC layer, the node matches the final destination address to its own address. If the packet is meant for it, the packet is handed over to the application layer. Else the data packet is forwarded to the next node towards the BS by sending it down to the MAC layer with the MAC address of the next hop node. If the packet received is a ROUTE_REQ type, a node is ascertaining the bidirectional link towards BS and the node will send a
ROUTE_ACK packet to the source node. If the packet type is ROUTE_ACK, the node is getting an acknowledgement from a node having route to BS and the routing table is updated with the source address of that packet as next hop node to the BS. In this protocol, the node keeps only one address towards BS and marks the state of the node to be having a route to the BS. The nodes having route to BS will start broadcasting ROUTE_BROADCAST packets periodically as stated above. If a node had requested route to a BS, and if the route cannot be established, it will get the ROUTE_FAIL message and the node drops the packet. Whenever the node obtains a route broadcast packet, the node will check the source address. If the address is that of the next hop address towards BS, the node reinitiates a timer to indicate validity of that route. If this timer expires before hearing a route broadcast from the next hop node, the route is deleted and the node reinvents a route to BS as described before. Meanwhile, the node stops its own route broadcasts also.

When a data packet is received from the application layer, it is to be sent to the next hop node to BS and send it down to the MAC layer. A timer de-activated on receiving the packet and activated after forwarding to the next hop will detect the break in streaming activity and enables the node to return to the non-streaming state to save power.

```cpp
void StreamingRoute::initialize(int stage)
{
    // To send route broadcast once a route to BS exists in the node.
    sendRoutePkt = new cMessage("sendRoutePkt");
    sendRoutePkt->setKind(ROUTE_BROADCAST);
    // To see if packets are arriving at the node at streaming rate.
    streamingTimer = new cMessage("streamingTimer");
    streamingTimer->setKind(STREAMING_TIMER);
    // If streaming is detected, publish it for other layers.
    streamingDataState = bb->subscribe(this, &streamData, -1);
}
```
void StreamingRoute::handleLowerMsg(cMessage* msg)
{
    if((finalDestAddr == myNetwAddr) && (m->getPktType() == DATA))
    {
        numPksReceived++;  // data collection in network layer
        sendUp(decapsMsg(m)); // send the packet to application.
    }
    else if (destAddr == myNetwAddr)
    {
        if(m->getPktType() == DATA)
        {
            // Make the packet for sink node with next hop address.
        }
        else if(m->getPktType() == ROUTE_REQ)
        {
            // Send ROUTE_ACK to source addr if route to BS exists.
        }
        else if(m->getPktType() == ROUTE_ACK)
        {
            // A route has been tested to be bidirectional. Update route table.
        }
        else if(m->getPktType() == ROUTE_FAIL)
        {
            delete m;
        }
    }
    else if (destAddr == bcastAddr)
    {
        // See the type. If route broadcast, then update the route table.
    }
    else if (finalDestAddr != sinkAddr)
    {
        // Get route. If not available, inform the source node about it.
    }
}

void StreamingRoute::handleUpperMsg(cMessage* msg)
{
    StreamingRoutePkt* netwPkt = new StreamingRoutePkt(msg->getName(), msg->getKind());
    netwPkt->setPktType(DATA);

    // updateRouteTable(finalDestAddr, netwPkt->getSrcAddr());
    // encapsulate the application packet
    assert(static_cast<cPacket*>(msg));
    netwPkt->encapsulate(static_cast<cPacket*>(msg));
    if (broadcastPkt)
        sendDown(netwPkt);
    else
    {
        drop(msg);
        delete msg;
        broadcastPkt = false;
    }

    scheduleAt(simTimeO + slotDuration*numSlots, streamingTimer);
}

Figure 5.20: Code segment in SOR Protocol implementation
128
The self message handling in the protocol is the heart of the implementation of the protocol. The ROUTE_BROADCAST is the message started by a timer in the node when it discovers a route to the BS. While handling this message, the node sends a ROUTE_BROADCAST packet to all nodes in its vicinity. This helps nodes to discover routes to BS as well as nodes to validate their existing routes to BS. The STREAMING_TIMER message will stop the streaming activity from that node, and the node will return to normal mode of operation to start the broadcast of route information. The updating of routing table is done when a data packet is received in a node from upper tier nodes on the way to BS. For getting a route to a node other than the BS, getRoute function is called in the protocol.

### 5.7 Simulation Results and Analysis of SOR Protocol

The SOR protocol was simulated with the parameters shown in Table 5.3. 40 nodes were deployed in a playground of size 1500m x 1500m with a maximum simulation time of 700s. The radio was set for carrier frequency of 2.445GHz with maximum transmit power of 0.5W. The receive sensitivity was -110dB. The nodes were randomly deployed in a distance of 60m to 900 m. Node 8 was selected as the SN to generate 500 packets at constant rate of one packet every 0.02s. The application was set to start after 500s of starting the simulation. The slot duration was set to 0.006s with 24 slot TDMA frame.

The setup delay of the routing protocol forms an important performance parameter for the protocol. The average setup delay (blue), the minimum setup delay (green) and the maximum setup delay (red) against tierNum is plotted in the Fig.5.21. As is obvious, the setup delay increases as the distance of the node from BS increases or the tier number increases. The maximum setup delay in the 6 tier network was around 94s. The per hop maximum delay was about 16s. Therefore, sufficient initialization time needs to be provided for nodes to find the routes towards BS. As is obvious, more the number of tiers, the required initialization time also increases. From multiple simulations with different node densities upto 120 nodes and 15 tiers, the per hop setup delay was found to be within 20s.
The SOR protocol was found to establish routes to the BS from the SN through multiple hops within a setup delay. These routes are found before the start of the application and maintained till they are active. Once a route is detected to be broken, the node finds a new one if there is any. The protocol was interfaced with suitable MAC protocol and found to offer the required throughput and delay performance for sustaining voice streaming in the network. SOR protocol could be interfaced with MLMAC protocol and the hosts could find the routes to the sink in a random deployment scenario for a suitable node density in the playground.
Hence, MLMAC-SOR was the recommended protocol suit for voice streaming in WSN during the initial part of this research work.

Figure 5.21: Route Setup Delay vs tierNum (Average, Min, Max)

5.8 Simulation Model for the DMPR Protocol

In DMPR Protocol, nodes keep multiple next hop addresses to BS so that the streaming can be distributed in the network topology. The routing protocol was named RouteMac in MF simulation model and is shown in Fig.5.22. A modified SOMAC protocol named RouteMAC was used in the hosts of this simulation. The modifications were to make the nodes in the listening range of streaming activity in the network to adopt the streaming slot cycle because the streaming could be switched through them by the higher tier node in the streaming path. The MAC layer in the simulation model is shown in Fig.5.23.

The main code segment of the protocol is shown in Fig.5.24. When ROUTE_ACK packets are received in the network layer, the new route is added to a queue. The size of the queue is decided by the node density in the network. The ROUTE_FLOOD packet received will help the node to add a new route to the BS by sending ROUTE_REQ if its source address is not in the routing table. If the received data packets are to be forwarded to BS, the node does a round robin
scheduling for forwarding the packets among all the nodes in the routing table.
Rest of the implementation remains same as the SOR implementation.

Figure 5.22: Network layer in the Hosts of DMPR protocol simulation

Figure 5.23: MAC layer in the Hosts of DMPR protocol simulation
void RouteMAC::handleLowerMsg(cMessage* msg)
{
    int macAddr;
    int bcastIpAddr = BaseMacLayer::NET_BROADCAST;
    RouteMACPkt* m = static_cast<RouteMACPkt*>(msg);
    int finalDestAddr = m->getFinalDestAddr();
    int destAddr = m->getDestAddr();
    int srcAddr = m->getSrcAddr();
    if((finalDestAddr==myNetwAddr) &&
        (m->getPktType()==DATA))
    {
        numPktsReceived++;
        sendUp(decapsMsg(m));
    }
    else if (destAddr == myNetwAddr)
    {
        if (m->getPktType()==DATA)
        {
            // Make the packet for sink node and return it.
            if (streamingTimer->isScheduled())
            {
                cancelEvent(streamingTimer);
                if (sendRoutePkt->isScheduled())
                    cancelEvent(sendRoutePkt);
            }
            else
            {
                // streamingState = true;
                numPktsReceived++;
                nbPktsToNxtHop++;
                if (countRoutes!=0)
                    if (nbPktsToNxtHop==5000/countRoutes)
                        { nbPktsToNxtHop = 0;
                           currNxtHopAddr++;
                           currNxtHopAddr = currNxtHopAddr % (countRoutes+1);
                           nextHopAddrToSink = routingTable[currNxtHopAddr];
                        }
                sendDown(encapsMsg(decapsMsg(m)));  
                scheduleAt(simTime()+slotDuration*numSlots, streamingTimer);
                updateRouteTable(finalDestAddr, srcAddr); // Route to the source of packet.
            }
        }
        else if (m->getPktType()==ROUTE_ACK)
        {
            // A route has been tested to be bidirectional. Update route table.
            if ((assuranceValue == 0) &&
                (routeReqState == true))
            {
                // A route has been tested to be bidirectional. Update route table.
```cpp
{ 
    assuranceValue = 1;
    nextHopAddrToSink = srcAddr;
    routingTable[0] = srcAddr;
    updateRouteTable (sinkIpAddr, srcAddr);
    routeSetupDelayToSink = simTime()-initializationTime;
    routeReqState = false;
}

else if((assuranceValue == 1) && (routeReqState == true))
{
    if(countRoutes <= 10)
    {
        countRoutes++;
        routingTable[countRoutes] = srcAddr;
        routeReqState = false;
    }
}
    delete m;
}

if(m->getPktType()==ROUTE_FLOOD)
{
    bool repeatAddr = false;
    for(int i=0; i<countRoutes; i++)
    {
        if (routingTable[i] == srcAddr)
            repeatAddr = true;
    }

    if(!repeatAddr)
    {
        RouteMACPkt *reqPkt = new RouteMACPkt("ROUTE_REQ",
ROUTE_REQ);
        reqPkt->setBitLength(headerLength);
        reqPkt->setDestAddr(m->getSrcAddr());
        reqPkt->setFinalDestAddr(m->getSrcAddr());
        reqPkt->setSrcAddr(myNetwAddr);
        reqPkt->setPktType(ROUTE_REQ);
        reqPkt->setAssuranceValue(assuranceValue);
        macAddr = arp->getMacAddr(m->getSrcAddr());
    }
}
```

Figure 24: Code segment in DMPR Protocol implementation

134
5.9 Simulation Results and Analysis of DMPR Protocol

The DMPR protocol was simulated with the parameters shown in Table 5.4. In a playground area of 100m x 1500m, 120 nodes were randomly deployed except the source and sink nodes which were designated as node 0 and node 8. The simulation time was set to 5000s. The radio was set to operate on carrier frequency 2.445GHz with maximum power 0.2W. The receiver sensitivity was set to -110dB. The SN was to generate 10000 data packets at a constant rate of one packet in every 0.018s with an initial delay of 4700s to let the network initialize. The MAC protocol was to operate with 0.06s slots and 64 slot TDMA frame.

Table 5.4: Simulation Parameters for DMPR Protocol

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>sim-time-limit</td>
<td>5000s</td>
</tr>
<tr>
<td>sim.playgroundSizeX</td>
<td>100</td>
</tr>
<tr>
<td>sim.playgroundSizeY</td>
<td>1500</td>
</tr>
<tr>
<td>sim.numHosts</td>
<td>120</td>
</tr>
<tr>
<td>sim.channelcontrol.carrierFrequency</td>
<td>2445e+6</td>
</tr>
<tr>
<td>sim.channelcontrol.pMax</td>
<td>0.20</td>
</tr>
<tr>
<td>sim.channelcontrol.sat</td>
<td>-110</td>
</tr>
<tr>
<td>sim.ROUTEMACHost[0].mobility.x</td>
<td>0</td>
</tr>
<tr>
<td>sim.ROUTEMACHost[0].mobility.y</td>
<td>0</td>
</tr>
<tr>
<td>sim.ROUTEMACHost[8].mobility.x</td>
<td>uniform(20, 100)</td>
</tr>
<tr>
<td>sim.ROUTEMACHost[8].mobility.y</td>
<td>uniform(1250, 1300)</td>
</tr>
<tr>
<td>sim.ROUTEMACHost[*].mobility.x</td>
<td>uniform(0,100)</td>
</tr>
<tr>
<td>sim.ROUTEMACHost[*].mobility.y</td>
<td>uniform (0,1400)</td>
</tr>
<tr>
<td>sim.ROUTEMACHost[8].appl.nbPackets</td>
<td>10000</td>
</tr>
<tr>
<td>sim.ROUTEMACHost[*].appl.nbPackets</td>
<td>0</td>
</tr>
<tr>
<td>sim.ROUTEMACHost[*].appl.trafficParam</td>
<td>0.018</td>
</tr>
<tr>
<td>sim.ROUTEMACHost[*].appl.initializationTime</td>
<td>4700s</td>
</tr>
<tr>
<td>sim.ROUTEMACHost[*].nic.mac.slotDuration</td>
<td>.06s</td>
</tr>
<tr>
<td>sim.ROUTEMACHost[*].nic.mac.controlSlotDuration</td>
<td>0.0005s</td>
</tr>
<tr>
<td>sim.ROUTEMACHost[*].nic.mac.subSlotDuration</td>
<td>0.006s</td>
</tr>
<tr>
<td>sim.ROUTEMACHost[*].nic.mac.numSlots</td>
<td>64</td>
</tr>
</tbody>
</table>
The DMPR protocol was simulated to establish the possibility of setting up multiple routes from the nodes to BS and the ability to distribute the streaming data packets among the routes in the node. With the simulation parameters in Table 5.4, a number of routes could be established from majority of the tiers as shown in Fig.5.25. Availability of multiple routes from middle tiers allow the streaming activity to be distributed across the topology of the network and helps in avoiding the hole formation within the network due to continued streaming for a long time. For example, from tier 9, there are 8 routes and the packets are distributed among them.

![Figure 5.25: Number of Routing Nodes vs tierNum](image)

The performance parameter of importance in routing is the route setup delay. Since this protocol is a proactive routing protocol, this delay decides the setup time required for the application. The minimum setup delay (blue), maximum setup delay (green) and average setup delay (red) vs tier number is shown in the Fig.5.26. As is obvious, setup delay increases as the tier number increases. It can
be observed that the maximum setup delay is 20s per hop. It helps in designing the setup time required in the application layer. For example, if the application envisages 20 hops between the SN and the BS based on the radio and terrain parameters, the application can start generating data packets after 400s which would be required for the network layer to setup the routes to BS with DMPR protocol.

More the number of routes from a node to BS, less the number of packets received by nodes in tier below that node and more the number of nodes in the streaming path in a tier, less the number of packets received by those nodes. It can be seen from Fig.5.27 that the middle tier nodes participate in the streaming activity handling less number of packets than the lower tier nodes. The distribution of packets among nodes in streaming is decided mainly by the topology of the network. It can be seen from the figure that there is distribution of streaming data packets in the middle region of the network topology, thereby reducing the chances of hole forming in the network.

5.10 Inference
The MAC and routing protocols developed initially established the possibility of voice streaming data rates in the WSN by stealing the transmission slots of a distributed TDMA MAC protocol. The MLMAC protocol was developed and tested with a fixed path routing algorithm in the network layer. It was used below the SOR protocol development in the MAC layer of the simulation. The combination of SOR-MLMAC protocols established the possibility of voice streaming data rates to be sustained in the WSN as the streaming path nodes were able to transmit in one slot every 3 slots thereby extracting a maximum of one third of the available data rate of the channel which is sufficient for voice streaming.
The slot size of the TDMA scheme used in the MLMAC protocol decides the energy efficiency of the nodes. Larger slot sizes are needed for better efficiency. For voice streaming applications, this would incur larger delays. In order to provide optimum energy efficiency and packet delay, SOMAC was developed based on MLMAC. In this protocol, streaming path nodes were adapting to a smaller slot size to reduce the streaming delay. The simulation of the protocol verified the scheme for providing voice streaming data rates. There were some initial packet losses during the streaming while the nodes were adapting to the streaming subslots which was a maximum of 10 packets per node. Therefore, with this protocol, the voice streaming application should cater for a setup delay.
The voice streaming being an energy consuming activity, single path streaming would have drained the batteries of the streaming path nodes, rendering them useless after a prolonged streaming activity. In order to avoid such hole formation in the network topology, DMPR protocol was developed in which the streaming activity gets distributed across the topology. The simulation showed the reduced amounts of packets handled by nodes in the streaming paths where multiple routes existed to BS. The DMPR-SOMAC combination therefore gives the optimum results in terms of geographical distribution of energy consumption and energy saving per node during streaming and nonstreaming states.