AUVNetSim: A Simulator for Underwater Acoustic Networks

Josep Miquel Jornet Montana
jmjornet@mit.edu

MIT Sea Grant College Program
Massachusetts Institute of Technology,
Cambridge, MA, 02139

Contents

Chapter 1

Installation

AUVNetSim is a simulation library for testing acoustic networking algorithms [1]. It is written in standard Python [2] and makes extensive use of the SimPy discrete event simulation package [3]. AUVNetSim is redistributed under the terms of the GNU General Public License.

AUVNetSim is interesting for both *end users* and *developers*. A user willing to run several simulations using the resources that are already available, can easily modify several system parameters without having to explicitly deal with Python code. A developer, who for example, wants to include a new MAC protocol, can simply do so by taking advantage of the existing structure.

1.1 Prerequisites

To run this software, the following packages are necessary before installing and running the AUVNetSim:

- Python Environment: the Python core software (release 2.5) [4].
- SimPy Package: a discrete-event simulation system [5].
- MatplotLib: a Python plotting library (release 0.91 or older) [6].
- Numpy: a package that provides scientific computing functionalities [7].

All of this software is freely available under the terms of the GNU license. Follow the instructions included within each package to properly complete their installation.

1.2 Installation

The AUVNetSim source code is available using subversion (SVN). There are several SVN clients for different platforms. For example, we use TortoiseSVN [8]. Once a tool like this is installed, a new user should:

- Create a new folder.
- Right click on the folder, SVN Checkout.
- Point to the AUVNetSim repository at [9].

Automatically, the contents of AUVNetSim will be downloaded to the local machine. The current stable version of the simulator is in the *trunk* subfolder. To install the simulator, just type `python setup.py` from the command line in that folder. To verify that the simulator has been properly installed, you can run the two examples that are included in the package.

Note that all the updates that are being done in the trunk subfolder can be directly seen by just updating the folder with the SVN tool. Users are only able to modify their local copy. If a major change is introduced, you can contact the project administrator and he will consider to either create a branch for you or to include the proposed features in the *trunk*.

Chapter 2

AUVNetSim for End Users

The simulator already contains a great variety of parameters and protocols for underwater acoustic networks that can be selected or modified. Rather than having to compile the code each time a new simulation is required, a user just needs to set up the simulation file (*.py) and the configuration file (*.conf).

2.1 Simulation File

This file contains the *main* function that will be invoked when the simulator is launched. The following Python code serves as an example:

```
import Simulation as AUVSim     # Inclusion of the resources
import pylab                    # Inclusion of the visualization class

def aSimulation():              # Main Function

    if(len(sys.argv) < 2):
        print "usage: ", sys.argv[0], "ConfigFile" # A configuration file is expected
    exit(1)

    config = AUVSim.ReadConfigFromFile(sys.argv[1])

    print "Running simulation"
    nodess = AUVSim.RunSimulation(config) # The simulation is launched
    print "Done"

    for x in nodess:
        nodes.append(nodess[x])

    PlotScenario3D(nodes)       # Visualization of the scenario for simulation
    PlotConsumption(nodes)      # Visualization of the consumption per node
    PlotDelay(nodes)            # Visualization of the delay per node
    pylab.show()

if __name__ == "__main__":
    aSimulation()
```

After the inclusion of the AUVNetSim library, the simulation is launched. Some of the results or statistics that are monitored throughout the simulation are displayed. The user can either use the already defined visualization functions, create new ones or save the required information in plain text files which later could be read using, for example, MATLAB. Several examples are included with the downloadable package.

2.2 Configuration File

Several parameters should be specified in the configuration file before each simulation. In the following lines, there is an example of the content of this type of files.

```
# Simulation Duration (seconds)
SimulationDuration = 3600.00

# Available Bandwidth (kHz)
BandWidth = 10.00

# Bandwidth efficiency (bps/Hz)
BandwidthBitrateRelation = 1.00

# Frequency (kHz)
Frequency = 10.00

# Maximum Transmit Power -> Acoustic Intensity (dB re uPa)
TransmitPower = 250.00

# Receive Power (dB) -> Battery Consumption (dB W)
ReceivePower = -10.00

# Listen Power (dB) -> Battery Consumption (dB W)
ListenPower = -30.00

# DataPacketLength (bits)
DataPacketLength = 9600.00 #bits

# PHY: set parameters for the physical layer
PHY = {"SIRThreshold": 15.00, "SNRThreshold": 20.00,"LISThreshold": 3.00, "variablePower": True,
"multicast2Distance": {0:1600.00,1:2300.00,2:2800.00,3:3200.00,5:6000.0}}

# MAC: define which protocol we are using & set parameters
MAC = {"protocol":"ALOHA", "max2resend":10.0, "attempts":4,"ACK_packet_length":24,
"RTS_packet_length":48, "CTS_packet_length":48,"WAR_packet_length":24, "SIL_packet_length":24,
"tmin/T":2.0,"twmin/T":0.0, "deltatdata":0.0, "deltad/T":0.0,}

# Routing: set parameters for the routing layer
Routing = {"Algorithm": "Static", "variation":2, "coneAngle":120.0}

# Nodes: here is where we define individual nodes
# Format: Address, Position [, period, destination]
Nodes = [["SinkA", (5000,5000,500), None, None],["SinkD",(5000,15000,500), None, None],
["SinkB", (15000,5000,500), None, None], ["SinkC", (15000,15000,500), None, None]]

# NodeField: Set up a field of nodes.
# Format (grid_block_size, N_blocks_wide, N_blocks_high[, bottom_left_corner[, node_ID_prefix])
NodeField = (5000.00, 4, 4, (0,0,0), "S")

# Maximum height/depth at which nodes can be
Height = 1000.00

# By default, the nodes in the node field are just relays, but we can make them
# also generate information by changing this value.
Period = 240.0

# It makes more sense to place nodes randomly rather than in a perfect grid
RandomGrid = True
```

```
# Nodes in the node field can be moving randomly and in without being
# aware of it because of underwater flows.
Moving = False
Speed = 0.0
```

All the possible parameters that can be currently specified are summarized in Table 2.1. The simulation can be started by just typing from the OS command line:

```
!> python simulation_file.py configuration_file.conf
```

In Appendix A, there is an introduction with some examples of how to externally control the simulator when using MATLAB. Despite not being the approach that we encourage, a user can just think of the simulator as a black box, with some inputs (the configuration file) and some outputs (which can be written in a text file). In this case, the user will rarely have to deal with Python code.

Physical Layer		
Center Frequency	[kHz]	
Bandwidth	[kHz]	
Bandwidth Efficiency	[bit/Hz]	
Transmitting mode max power	[dBreμPa]	
Receiving power consumption	[dBW]	
Listening power consumption	[dBW]	
Listening threshold	[dB]	
Receiving threshold	[dB]	
Power Control	True/False	
Power Levels	name:value[km]	Only if Power Control is used
Medium Access Control		
Protocol	CS-ALOHA, DACAP, CSMA	
RTS length	[bit]	Only if DACAP is used
CTS length	[bit]	Only if DACAP is used
ACK length	[bit]	
WAR length	[bit]	Only if DACAP is used
SIL length	[bit]	Only if DACAP is used
DATA length	[bit]	
Retransmission attempts		
Maximum waiting time	[s]	Only if ALOHA is used
Tmin		Only if DACAP is used
Twmin		Only if DACAP is used
Interference region		Only if DACAP is used
Routing Layer		
Protocol	No routes, Static Routes, FBR	
Variation	0,1,2	Only when static routes are used
Cone aperture	[degree]	Only if FBR is used
Retransmission attempts		Only if FBR is used
Nodes		
Name		
Period	[s]	Can be None
Destination	[node]	Can be None
Position or Path	List of points	
Random Position	True/False	
Random Moving	True/False	
Moving Speed	[m/s]	
Simulation Duration	[s]	

Table 2.1: AUVNetSim parameters that should be specified in the configuration file

Chapter 3

AUVNetSim for Developers

Before reading this section, we encourage the user to familiarize with the Python programming language and the SimPy library [2, 3].

The way in which AUVNetSim is programmed eases the task of including new features. Like many other wireless network simulators, the description of the different layers functionalities is specified in different classes or files. A programmer willing to introduce, for example, a new routing technique does not need deal with the MAC or the physical layer.

The communication between layers is performed by the exchange of short messages. For example, a packet coming from the application layer is sent to the routing layer, which will update the packet header and, on its turn, will send it to the MAC layer. Finally, the message will be transmitted to the channel through the physical layer, following the protocol policy.

In the following lines, an overview of each of the files that compose the simulator is offered.

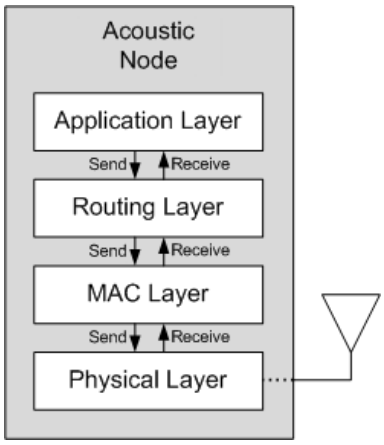

Figure 3.1: AUVNetSim: node programming structure.

3.1 Simulation.py

This is the main file for a project. In here, the 3D scenario for simulation is created and the simulation is conducted.

```
import SimPy.Simulation as Sim          # Inclusion of the discrete-event mechanism
from AcousticNode import AcousticNode   # Contains the definition of a node

def RunSimulation(config_dict):
    Sim.initialize()
```

```
    # This is the initial event that will trigger the simulation
    AcousticEvent = Sim.SimEvent("AcousticEvent")

    # Nodes are set up according to config_dict, which is the content of the configuration file.
    nodes = SetupNodesForSimulation(config_dict, AcousticEvent)

    # The simulation is run until the end of events or for the maximum duration time
    Sim.simulate(until=config_dict["SimulationDuration"])

    return nodes
```

A scenario is defined according to the description in the configuration file. There are two ways of specifying the nodes that the system contains:

- Each node can be specified by its name and position (or a path if it is a mobile node). If it is an active node, the packet generation rate and the packets' destination (a new destination can be randomly chosen before each transmission) should also be included.
- A node-field can be defined by the 3D region that it is covering and the number of nodes that are positioned in it. Nodes can be completely randomly positioned or randomly positioned within a grid. The nodes in a node-field are usually just relays, but it is easy to make them generate information too. In addition, involuntary drifting can also be modeled, just indicating that they are moving and specifying their speed.

```
def SetupNodesForSimulation(config_dict, acoustic_event):
    nodes = [ ]

    # Single nodes
    if "Nodes" in config_dict.keys():
        for n in config_dict["Nodes"]:
            cn = [config_dict,] + n
            nodes.append(AcousticNode(acoustic_event, *cn))

    # Node field
    if "NodeField" in config_dict.keys():
        nodes += CreateRandomNodeField(acoustic_event, config_dict,*config_dict["NodeField"])

    return nodes
```

The way in which the nodes in the relay field are involuntary moving is shown below. When initialized, four random points are created, and following the specified speed, they will travel along them.

```
def CreateRandomNodeField(acoustic_event, config, box_len, nWide, nHigh,
                          bottom_left_position=(0,0,0), Prefix="StaticNode"):
    import random
    random.seed()
    node_positions=[ ]

    # Nodes can be randomly located within a virtual grid and moving randomly
    if not config["RandomGrid"] and not config["Moving"]:
        [node_positions.append((bottom_left_position[0]+box_len*x,
                                bottom_left_position[1]+box_len*y,
                                bottom_left_position[2])) for x in range(nWide) for y in range(nHigh)]
    elif config["RandomGrid"] and not config["Moving"]:
        [node_positions.append((bottom_left_position[0]+box_len*random.random()+box_len*x,
                                bottom_left_position[1]+box_len*random.random()+box_len*y,
                                bottom_left_position[2]+config["Height"]*random.random()))
                                for x in range(nWide) for y in range(nHigh)]
```

```
    else:
        [node_positions.append(([(bottom_left_position[0]+box_len*random.random()+box_len*x,
                                  bottom_left_position[1]+box_len*random.random()+box_len*y,
                                  bottom_left_position[2]+config["Height"]*random.random()),
                                 (bottom_left_position[0]+box_len*random.random()+box_len*x,
                                  bottom_left_position[1]+box_len*random.random()+box_len*y,
                                  bottom_left_position[2]+config["Height"]*random.random()),
                                 (bottom_left_position[0]+box_len*random.random()+box_len*x,
                                  bottom_left_position[1]+box_len*random.random()+box_len*y,
                                  bottom_left_position[2]+config["Height"]*random.random()),
                                 (bottom_left_position[0]+box_len*random.random()+box_len*x,
                                  bottom_left_position[1]+box_len*random.random()+box_len*y,
                                  bottom_left_position[2]+config["Height"]*random.random())],
                                  config["Speed"]) ) for x in range(nWide) for y in range(nHigh)]
    nodes={}
    j = enumerate(node_positions)
    for num, pos in j:
        name = "%s%03d" % (Prefix,num+1)
        nodes[name]=AcousticNode(acoustic_event, config, name, pos, config["Period"],
                                  None, config["Moving"])
    return nodes
```

3.2 AcousticNode.py

This file contains the description of an acoustic node. Within this class, the different functionalities of a single node are initialized according to the configuration file. A node is determined by:

- Name and position.
- Characteristics of its physical layer.
- Medium Access Control protocol in use.
- Routing technique.
- Packet Generation Rate and packets' destination.

```
class AcousticNode():
    def __init__(self, event, config, label, position_or_path, period=None,
                 destination=None, involuntary_moving=False):
        """Initialization of the acoustic node.
        """
        self.config = config
        self.name = label
        self.SetupPath(position_or_path)
        self.involuntary_moving = involuntary_moving

        # Physical layer
        self.physical_layer = PhysicalLayer(self, config["PHY"], event)

        # MAC Layer
        self.MACProtocol = SetupMAC(self, config["MAC"])

        # Routing Layer
        self.routing_layer = SetupRouting(self, config["Routing"])

        # Application Layer
        self.app_layer = ApplicationLayer(self)
        if(period is not None):
            self.SetUpPeriodicTransmission(period, destination)
```

Nodes mobility is addressed as follows. A mobile node is defined as a set of points and the speed at which it travels along them. From the path class, the specific position at a time can be obtained. Taking into account that nodes can be moving both being aware or without knowing it, we define two different functions:

```
def GetCurrentPosition(self):
    if not self.involuntary_moving:
        # Returns the real position of the nodes, which are not involuntary moving
        return self.path.get_position_at_time(Sim.now())
    else:
        # Returns the initial position, the only one that nodes may know if they are involuntary moving
        return self.path.get_initial_position()

def GetRealCurrentPosition(self):
    # Returns the real position, which may show the effect of involuntary drifting if that's the case
    return self.path.get_position_at_time(Sim.now())
```

3.3 PhysicalLayer.py

The physical layer is the key file of the entire simulator. The correctness of its content will affect the entire simulation. This is why an extra level of detail is given in this section. The physical layer models both, each user's modem and transducer, and all users' common channel. This file contains the following Python classes.

Several system performance parameters are measured at this level, including the energy consumed in transmissions, the energy consumed in listening to the channel, and the number of collisions detected.

The channel model is also contained in this file. A packet being transmitted will be delayed according to acoustic propagation and its power will be attenuated according to the acoustic path-loss model defined in the same file.

A developer can easily introduce new channel models, variables to monitor, modem functionalities, but there are two functions that should be always preserved. These are the ones that are used to communicate from and to the layer immediately above, in this case, the MAC layer.

```
def TransmitPacket(self, packet):
    ''' Function called from the upper layers to transmit a packet.
    '''
    # It is MAC protocol duty to check before transmitting if the channel is idle
    # using the IsIdle() function.
    if self.IsIdle()==False:
        self.PrintMessage("I should not do this ... the channel was not idle!")

    self.collision = False # Initializing the flag
    if self.variable_power:
        distance = self.multicast2distance[packet["level"]]
        power = distance2Intensity(self.bandwidth, self.freq,
                                   distance, self.SNR_threshold)
    else:
        power = self.transmit_power # Default maximum power

    new_transmission = OutgoingPacket(self)
    Sim.activate(new_transmission, new_transmission.transmit(packet, power))

def OnSuccessfulReceipt(self, packet):
    ''' Function called from the lower layers when a packet is received.
    '''
    self.node.MACProtocol.OnNewPacket(packet)
```

3.3.1 Physical Layer

This is the class that allows the interaction between this layer, the MAC protocol and other layers above (e.g., when thinking of a cross-layer design). In here, all the parameters specified in the configuration file

regarding the physical layer are initialized. Several system performance parameters are measured at this level, including the energy consumed in transmissions, the energy consumed in listening to the channel, and the number of collisions detected.

The physical layer does not *think*, i.e., if the MAC layer asks this layer to transmit a packet, it will simply do it, even if the channel is busy. The channel status can be checked from the layers above to obtain valuable information such as:

- If the channel is idle or not (this should be actually checked before transmitting).
- If there has recently been a collision (may be interesting after waiting for two times the maximum transmission time, for example).

When the MAC layer wants to transmit a packet, it will invoke the transmitting function:

```
# Function called from the layers above. The MAC layer needs to transmit a packet
def TransmitPacket(self, packet):

    if self.IsIdle()==False:
        # The MAC protocol is the one that should check this before transmitting
        self.PrintMessage("I should not do this... the channel is not idle!")

    self.collision = False

    if self.variable_power:
        distance = self.level2distance[packet["level"]]
        power = distance2Intensity(self.bandwidth, self.freq, distance, self.SNR_threshold)
    else:
        power = self.transmit_power

    new_transmission = OutgoingPacket(self)
    Sim.activate(new_transmission, new_transmission.transmit(packet, power))
```

When a packet is properly received, this is passed to the MAC layer:

```
# When a packet has been received, we should pass it to the MAC layer
def OnSuccessfulReceipt(self, packet):
    self.node.MACProtocol.OnNewPacket(packet)
```

3.3.2 Outgoing Packet

In the physical layer, a transmitted packet is modeled using the `OutgoingPacket` class. As shown in the following lines, an outgoing packet will occupy the node's modem for the transmission time, which is obtained accordingly to the packet length and the modem bit-rate (this is obtained from the bandwidth that is being used and the bandwidth efficiency, both specified in the configuration file). At the same time, an event will be triggered in all the nodes in the network, accounting for the packet reception. All this is done in the transmit function in the `OutgoingPacket` class.

```
def transmit(self, packet, power):
    # Hold onto the modem in order to model the half-duplex operation of underwater acoustic modems.
    yield Sim.request, self, self.physical_layer.modem

    # Real bit-rate
    bandwidth = self.physical_layer.bandwidth
    bitrate = bandwidth*1e3*self.physical_layer.band2bit
    duration = packet["length"]/bitrate

    self.physical_layer.transducer.channel_event.signal(
                    {"pos": self.physical_layer.node.GetRealCurrentPosition(),
                     "power": power, "duration": duration, "frequency": self.physical_layer.freq,
                     "packet": packet})
```

```
    # Hold onto the transducer for the duration of the transmission
    self.physical_layer.transducer.OnTransmitBegin()
    yield Sim.hold, self, duration
    self.physical_layer.transducer.OnTransmitComplete()

    # Release the modem when done
    yield Sim.release, self, self.physical_layer.modem

    # This is used to take statics about energy consumption
    power_w = DB2Linear(AcousticPower(power))
    self.physical_layer.tx_energy += (power_w * duration)
```

3.3.3 Arrival Scheduler

In order to program an event regarding the reception of the packet at each node, we define the `ArrivalScheduler` class. At each node, this will calculate the time at which the packet reception will start and the power that the packet will have. This is done using the real position of the nodes (the one that takes into account involuntary movements), according to the acoustic waves propagation speed and modeling the underwater acoustic path-loss using Thorp's expression for acoustic absorption.

```
def schedule_arrival(self, transducer, params, pos):
    distance = pos.distanceto(params["pos"])

    if distance > 0.01:  # I should not receive my own transmissions
        receive_power = params["power"] - Attenuation(params["frequency"], distance)
        travel_time = distance/1482.0  # Speed of sound in water = 1482.0 m/s

        yield Sim.hold, self, travel_time

        new_incoming_packet = IncomingPacket(DB2Linear(receive_power),
                                             params["packet"], transducer.physical_layer)
        Sim.activate(new_incoming_packet,new_incoming_packet.Receive(params["duration"]))
```

3.3.4 Incoming Packet

The reception of a packet is modeled using the `IncomingPacket` class. In this, a part from the user content, information regarding its own status in terms of interference is also included, i.e., the Signal-to-Noise Ratio or SIR is monitored during the packet's life. When an incoming packet is being received, it will hold the user's transducer for the reception time.

```
def UpdateInterference(self, interference):
    if(interference > self.MaxInterference):
        self.MaxInterference = interference

def Receive(self, duration):
    if self.power >= self.physical_layer.listening_threshold:
       # Otherwise I will not even notice that there are packets in the network
       yield Sim.request, self, self.physical_layer.transducer
       yield Sim.hold, self, duration
       yield Sim.release, self, self.physical_layer.transducer

       # Even if a packet is not received properly, we have consumed power
       self.physical_layer.rx_energy += DB2Linear(self.physical_layer.receive_power) * duration

def GetMinSIR(self):
    return self.power/(self.MaxInterference-self.power)
```

3.3.5 The Transducer

This class is used to model each user's transducer. The transducer can be seen as a list of incoming and outgoing packets. When a new packet arrives (an `IncomingPacket`), all the packets in the list, which are those being concurrently received, will update their SIR. Remember that the `ArrivalScheduler` has previously computed the power of each packet at each destination. At the same time, note that if the MAC layer decides to transmit a packet (an `OutgoingPacket`) even having the transducer not empty, taking into account the half-duplex operation of underwater acoustic modems, all the packets that are being received will be discarded or doomed.

```
# Override SimPy Resource's "_request" function to update SIR for all incoming messages.
def _request(self, arg):
    # We should update the activeQ
    Sim.Resource._request(self, arg)

    # "arg[1] is a reference to the IncomingPacket instance that has been just created
    new_packet = arg[1]

    # Doom any newly incoming packets to failure if the transducer is transmitting
    if self.transmitting:
        new_packet.Doom()

    # Update all incoming packets' SIR
    self.interference += new_packet.power

    [i.UpdateInterference(self.interference, new_packet.packet) for i in self.activeQ]
```

When the receiving time has been completed, a packet will either be: properly received (enough power, enough SIR), discarded due to interference because of a collision (enough power, not enough SIR) or discarded (not enough power).

```
# Override SimPy Resource's "_release" function to update SIR for all incoming messages.
def _release(self, arg):
    # "arg[1] is a reference to the IncomingPacket instance that just completed
    doomed = arg[1].doomed
    minSIR = arg[1].GetMinSIR()
    new_packet = deepcopy(arg[1].packet)

    # Reduce the overall interference by this message's power
    self.interference -= arg[1].power

    # Delete this from the transducer queue by calling the Parent form of "_release"
    Sim.Resource._release(self, arg)

    if not doomed and Linear2DB(minSIR) >= self.SIR_thresh
        and arg[1].power >= self.physical_layer.receiving_threshold:

        # Properly received: enough power, not enough interference
        self.collision = False
        self.on_success(new_packet)

    elif arg[1].power >= self.physical_layer.receiving_threshold:
         # Too much interference but enough power to receive it: it suffered a collision
         if self.physical_layer.node.name == new_packet["through"]
            or self.physical_layer.node.name == new_packet["dest"]:

             self.collision = True
             self.collisions.append(new_packet)
             self.physical_layer.PrintMessage("A "+new_packet["type"]+" packet to "
```

```
                                        + new_packet["through"]
                                        +" was discarded due to interference.")
    else:
        # Not enough power to be properly received: just heard.
        self.physical_layer.PrintMessage("This packet was not addressed to me.")
```

Finally, note that the physical layer does not care about addressing. All packets that are being received will be transmitted to the MAC layer, who again, is the only one that makes decisions (accepting a packet and reacting accordingly, or just discarding it).

3.4 MAC.py

The MAC protocol is the one able to *command* the physical layer. As noted before, it is its responsibility to check if the channel is idle or not before transmitting, to retransmit if necessary and after a given time, or to consider or not a packet that has been received, amongst others. At the same time, it will interact with the layers above, i.e., the routing layer.

Any MAC protocol can be described as a set of rules. Different MAC protocols are defined in this file. CS-ALOHA, DACAP and CSMA and their adoptions for the FBR protocol are already included in the library. It is not the aim of this document to explain the way in which these are implemented. As in the previous case, there are some functions that should be always preserved:

```
def InitiateTransmission(self, OutgoingPacket):
    ''' Function called from the upper layers to transmit a packet.
    '''

    self.outgoing_packet_queue.append(OutgoingPacket)
    self.fsm.process("send_data")


def OnNewPacket(self, IncomingPacket):
    ''' Function called from the lower layers when a packet is received.
    '''
    self.incoming_packet = IncomingPacket
    if self.IsForMe():
        self.fsm.process(self.packet_signal[IncomingPacket["type"]])
    else:
        self.OverHearing()
```

The `FSM.py` class is used to implement Finite State Machines (common among MAC protocols). Any state diagram can be easily reproduced by defining the different states and all the possible transitions between them.

It is also common in MAC protocols to make use of timers to schedule waiting or back-off periods. A timer will trigger an event if it is not stopped once the time is consumed.

```
class InternalTimer(Sim.Process):
    def __init__(self, fsm):
        Sim.Process.__init__(self, name="MAC_Timer")
        random.seed()
        self.fsm = fsm

    def Lifecycle(self, Request):
        while True:
            yield Sim.waitevent, self, Request
            yield Sim.hold, self, Request.signalparam[0]
            if(self.interrupted()):
                # Just ignores the time finalization
                self.interruptReset()
```

```
            else:
                # Triggers a new transition in the state diagram
                self.fsm.process(Request.signalparam[1])
```

3.5 Routing Layer

The routing layer contains a description of different routing techniques. Right now, Dijkstra's algorithm can be used to obtain minimum power routes as well as the novel Focused Beam Routing protocol. As in the previous layers, independently of the protocol, the functions that interact with the MAC protocol and the application layer should be preserved:

```
class SimpleRoutingTable(dict):

    def SendPacket(self, packet):
        packet["level"]=0.0
        packet["route"].append((self.node.name, self.node.GetCurrentPosition()))
        try:
            packet["through"] = self[packet["dest"]]
        except KeyError:
            packet["through"] = packet["dest"]

        self.node.MACProtocol.InitiateTransmission(packet)

    def OnPacketReception(self, packet):
        # If this is the final destination of the packet,
        # pass it to the application layer
        # otherwise, send it on...
        if packet["dest"] == self.node.name:
            self.node.app_layer.OnPacketReception(packet)
        else:
            SendPacket(packet)
```

Note that as the coupling between the MAC protocol and the routing technique increases, there are more functionalities cross-referenced between classes.

3.6 Application Layer

In the application layer, packets may be periodically generated and relayed to the lower layers. Different information flows can be modeled. Right now, a Poisson information source is used, but this can be easily changed. In addition, some system performance parameters are monitored such as the packet end-to-end delay or the number of hops that a packet has made before reaching its final destination.

```
def PeriodicTransmission(self, period, destination):
    while True:
        self.packets_sent+=1
        packet_ID =  self.node.name+str(self.packets_sent)

        if destination==None:
             destination = "AnySink"

        packet = {"ID": packet_ID, "dest": destination, "source": self.node.name, "route": [ ],
                  "type": "DATA", "initial_time": Sim.now(),
                  "length":self.node.config["DataPacketLength"]}
        self.node.routing_layer.SendPacket(packet)
        next = poisson(period)
        yield Sim.hold, self, next
```

```
def OnPacketReception(self, packet):
    self.log.append(packet)
    origin = packet["route"][0][0]
    if origin in self.packets_received.keys():
        self.packets_received[origin]+=1
    else:
        self.packets_received[origin]=1

    delay = Sim.now()-packet["initial_time"]
    hops = len(packet["route"])

    self.PrintMessage("Packet "+packet["ID"]+" received over "+str(hops)+
                      " hops with a delay of "+str(delay)+
                      "s (delay/hop="+str(delay/hops)+").")

    self.packets_time.append(delay)
    self.packets_hops.append(hops)
    self.packets_dhops.append(delay/hops)
```

3.7 Visualization Functionalities

Last but not least, there are several functions that are included in the downloadable package that can be used to illustrate the results and check at a glance the system overall performance. All them make an extensive use of the MatPlotLib/Pylab package and can be found in the Sim.py file. The main idea is to process the different variables that are monitored during the simulation, from the structure containing all the nodes.

In Fig.3.2, the scenario for a simulation containing four active nodes (A, B, C, D), transmitting to a common sink, and 64 relays is shown. Within this graph, we are able to show plenty of information:

1. The color of a node is related to the energy that it has consumed by just listening to the channel. That is to say, a node surrounded by active nodes will have a red-like color.

2. The size of a node is proportional to the energy that it has consumed transmitting packets in the network. A bigger node has taken part in more packet exchanges than a smaller node.

3. Routes can then visually be identified.

Alternatively, the energy consumed in both transmitting and receiving packets can be plotted in a bar diagram, such as the one shown in Fig.3.3. Fig.3.4 contains a histogram of the end-to-end delay of the packets received at the common sink. 3D graphs are also possible. For example, in Fig.3.5 a network containing an AUV and a node-field with 16 relays is shown. In the same plot, the route that each packet has followed is included.

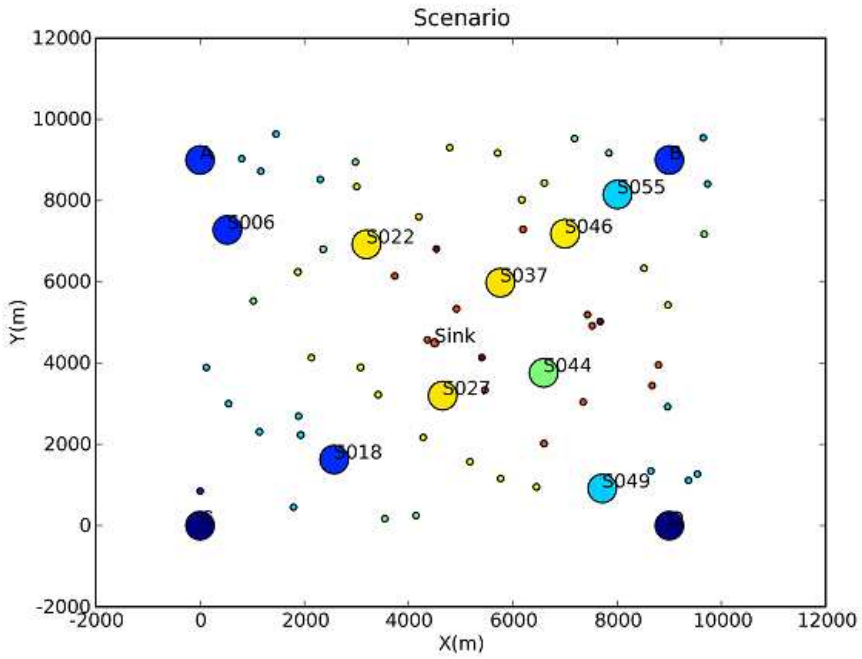

Figure 3.2: AUVNetSim: scenario for simulation and final node status.

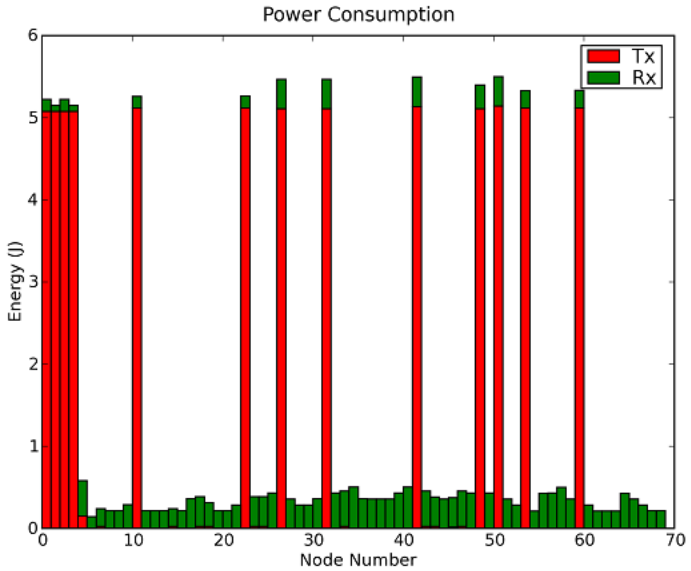

Figure 3.3: AUVNetSim: energy consumption at each node of the scenario.

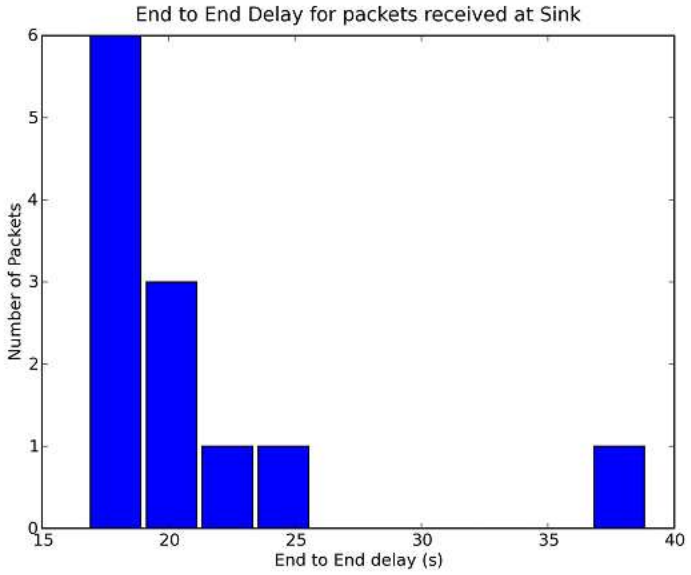

Figure 3.4: AUVNetSim: histogram of the end-to-end delay of the packets received at the common sink.

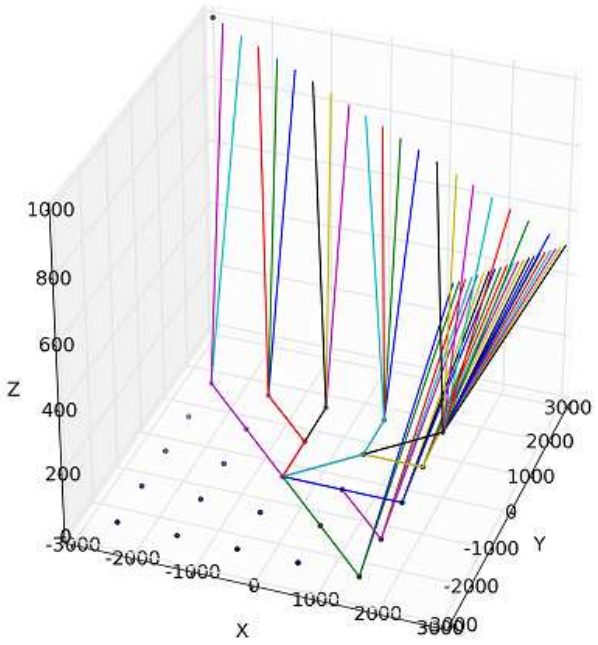

Figure 3.5: AUVNetSim: 3D scenario containing an AUV and a node-field with 16 relays.

Appendix A

Using AUVNetSim with MATLAB

From an end-user perspective, the AUVNetSim can be seen as a black box, with some inputs (the configuration file) and some outputs. These outputs or results, rather than just being printed on the screen, can be also saved in a text file, as shown in one of the simulation files included in the simulator package.

Thus, the simulator can be *externally* controlled, just writing the appropriate values on the configuration file; and *monitored*, reading the results from a text file. This simplifies even more the use of the simulator for new end-users, which will not have to deal with Python at all (despite we do not encourage this approach). An example of how to invoke the simulator from MATLAB and process its results is included in the following lines.

A.1 The main script

In the main MATLAB script, we will have to define the simulation parameters, write the configuration file, launch the simulation and show the results. The file is self-contained, please read the comments.

```
% % Sample file for externally governing the AUVNetSim
clear all % Clears all variables in the system memory
close all % Close all existing windows
clc       % Cleans the command window content

% The parameters are defined as follows:

% nl = number of transmission power levels when using power control.
nn = 1;

for n = (8) % Different node densities can be evaluated creating a loop
    % n = grid-cell positions is one dimension. N=n.^2 stands for the
    % number of nodes.

    % Scenario
    if nn ~= 1
        clear all
        load status.mat
    end

    c = 20e3; % The length of the side over which nodes are deployed (Total area = c.^2)
    h = 1e3; % Maximum difference in height between nodes
    rho = n.^2./c.^2; % The node density

    randomgrid = 'True'; % Scenario: are nodes randomly deployed?
    moving = 'True'; % Scenario: are nodes randomly moving?
    speed = 0.2; % If so, at which speed?
```

```
MAC = 'DACAP4FBR'; % MAC protocol that will be used
ROU = 'FBR'; % Routing technique that will be used
theta = 120.0; % Only if using the FBR protocol
variation = 0; % and its variation
datalength = 9600; % Bits
period = 30; % Seconds

simtime = 60*60; % Seconds

save scenario.mat c h n rho

for nl = (4) % Different number of levels can be evaluated sequentially
    % For each node density and/or number of power levels, different
    % center frequency and bandwidth may be chosen
    % selfreq % NOT included within the package

    % Or not, we may want to always use the same ones
    fc = 20; % kHz
    b = 1; % kHz

    % We may want to define different power levels using different
    % criteria.
    setlevels

    load scenario.mat

    % We should save the name of the file containing the results from different
    % simulations, in order to load it afterwards
    if n<10
        if fc<10
            name(nn,:) = strcat('sim_N_',num2str(n,'%2g'),'_f_',num2str(fc,'%6.2f'),
                                '_b_',num2str(b,'%6.2f'),'_',MAC,'_',ROU,'.txt');
        else
            name(nn,:) = strcat('sim_N_',num2str(n,'%2g'),'_f_',num2str(fc,'%6.2f'),
                                '_b_',num2str(b,'%6.2f'),'_',MAC,'_',ROU,'.tx');
        end
    else

        if fc<10
            name(nn,:) = strcat('sim_N_',num2str(n,'%2g'),'_f_',num2str(fc,'%6.2f'),
                                '_b_',num2str(b,'%6.2f'),'_',MAC,'_',ROU,'.tx');
        else
            name(nn,:) = strcat('sim_N_',num2str(n,'%2g'),'_f_',num2str(fc,'%6.2f'),
                                '_b_',num2str(b,'%6.2f'),'_',MAC,'_',ROU,'.t');
        end
    end

    % The configuration file is written
    write_conf

    % The simulator is launched
    !python Sim.py config.conf

    nn=nn+1;
end

n=n+1;
save status.mat name n nn
```

```
end

% Example script in which the scenario and the routes that different
% packets have followed is included.
plotroutes

% Example script in which the average end-to-end delay, the number of
% collisions an the energy per bit consumed is shown
show_results
```

A.2 Writing the configuration file

The configuration file should specify all the simulation parameters. Some of them have been defined in the main file, all the others can be changed from here.

```
% % Writes the configuration file

% Multi refers to the power levels, it is defined as explained in the
% manual: level_name:distance_to_cover.
multi=strcat('0:',num2str(levels(1),'%6.2f'));
for j=1:(nl-1)
    multi=strcat(multi,',',num2str(j,'%2g'),':',num2str(levels(j+1),'%6.2f'));
end

% We use this to create the configuration file for the AUVNetSimulator
fid = fopen('config.conf','wt');
fprintf(fid,'# AUTOGENERATED Config File for AUVNetSim\n');
fprintf(fid,'# Simulation Duration (seconds)\n');
fprintf(fid,'SimulationDuration = %6.2f \n\n', simtime);
fprintf(fid,'# Available Bandwidth (kHz) \n');
fprintf(fid,'BandWidth = %6.2f \n\n', b);
fprintf(fid,'# Bandwidth and bitrate relation (bps/Hz) \n');
fprintf(fid,'BandwidthBitrateRelation = %6.2f \n\n', 1);
fprintf(fid,'# Frequency (kHz) \n');
fprintf(fid,'Frequency = %6.2f \n\n', fc);
fprintf(fid,'# Transmit Power
fprintf(fid,'TransmitPower = %6.2f \n\n', 250.0);
fprintf(fid,'# Receive Power (dB)
fprintf(fid,'ReceivePower = %6.2f \n\n',-30.0);
fprintf(fid,'# Listen Power (dB)
fprintf(fid,'ListenPower = %6.2f \n\n',-30.0);
fprintf(fid,'# DataPacketLength (bits) \n');
fprintf(fid,'DataPacketLength = %6.2f #bits \n\n', datalength);
fprintf(fid,'# PHY: set parameters for the physical layer \n');
fprintf(fid,'PHY = {"SIRThreshold":%6.2f, "SNRThreshold":%6.2f, "LISThreshold":%6.2f,
            "variablePower":True,"variableBandwidth":False,"level2distance":{',15.0,20.0,3.0);
fprintf(fid,multi);
fprintf(fid,'}}\n\n');
fprintf(fid,'# MAC: define which protocol we are using & set params\n');
fprintf(fid,'MAC = {"protocol":"%s", "max2resend":10.0, "attempts":4,
            "ACK_packet_length":24, "RTS_packet_length":48, "CTS_packet_length":48,
            "WAR_packet_length":24, "SIL_packet_length":24, "tmin/T":2.0, "twmin/T":0.0,
            "deltatdata":0.0, "deltad/T":0.0, }\n\n',MAC);
fprintf(fid,'# Routing: set parameters for the routing layer\n');
fprintf(fid,'Routing = {"Algorithm": "%s", "variation":%g, "coneAngle":%2g, "coneRadius":%6.2f,
            "maxDistance":10e3}\n\n',ROU,variation,theta,levels(1)-50.0); %#ok<LTARG>
fprintf(fid,'# Nodes: here is where we define individual nodes\n');
fprintf(fid,'# format: AcousticNode(Address, position[, period, destination])\n');
```

```
fprintf(fid,'Nodes = [["SinkA", (5000,5000,500), None, None],["SinkD",(5000,15000,500), None, None],
            ["SinkB", (15000,5000,500), None, None], ["SinkC", (15000,15000,500), None, None]]\n\n');
fprintf(fid,'# NodeField: Set up a field of nodes\n');
fprintf(fid,'# format (grid_block_size, N_blocks_wide, N_blocks_high[,
             bottom_left_corner[, node_ID_prefix])\n');
fprintf(fid,'NodeField = (%6.2f,%g,%g, (0,0,0), "S")\n\n',c./n,n,n);
fprintf(fid,'Period=%6.1f\n',period);
fprintf(fid,'RandomGrid=%s\n',randomgrid);
fprintf(fid,'Moving=%s\n',moving);
fprintf(fid,'Speed=%6.2f\n',speed);
fprintf(fid,'Height=%6.2f',h);
fclose(fid);
```

A.3 Showing the results

Different results can be obtained from the simulator. For example, it may be interesting to replicate the scenario that is plotted when using the PlotScenario3D function within the visualization functions. This is what is done using the MATLAB script plotroutes.m:

```
% The simulator stores the positions of nodes in a file like this one
positions = importfile('Positions_DACAP4FBR_0.txt');

% Size = transmission energy, Color = receiving energy
scatter3(positions(:,1),positions(:,2),positions(:,3),round(positions(:,4)./max(positions(:,4))*1000+1),
        round(positions(:,5)./max(positions(:,5))*1000+1),'filled')

xlim([0 c])
ylim([0 c])
zlim([0 h])

set(gca,'XTick',0:c/(n-1):c)
set(gca,'YTick',0:c/(n-1):c)
set(gca,'ZTick',0:h:h)

% The simulator stores the positions of nodes in a file like this one
routes=importfile1('Routes_DACAP4FBR_0.txt');

% Now we can just plot the routes on them
hold on

for i=2:length(routes(:,1))
    x=[ ];
    y=[ ];
    z=[ ];
    for j=1:3:length(routes(1,:))
        if routes(i,j)==-1
            break
        end
        x=[x, routes(i,j)];
        y=[y, routes(i,j+1)];
        z=[z, routes(i,j+2)];
    end
    plot3(x,y,z)
end

axis equal
xlabel('x [m]')
ylabel('y [m]')
zlabel('z [m]')
```

When the simulation has been running for different node densities, it may also be interesting to show the system performance as a function of the node density. For example, the energy per bit consumption, the number of collisions and the average packet end-to-end delay is shown in the following lines:

```
% This only makes sense when the simulation has been run sequentially
% for different node densities

n0=4; % Initial number of nodes in one dimension
nf=16; % Final number of nodes in one dimension

% Note that all the rows in a matrix should have the same length, this is
% why we have to start doing this concatenating stuff...
for nnn=1:1
    results(:,:,nnn) = importfile(strcat(name(nnn,:),'t'));
end

for nnn=2:6
    results(:,:,nnn) = importfile(strcat(name(nnn,:),'xt'));
end

for nnn=7:13
    results(:,:,nnn) = importfile(strcat(name(nnn,:),'txt'));
end

figure()
% In the third column of the matrix, we have the information regarding the
% energy consumed in the system.
% In the seventh column of the matrix, we have the information regarding
% the number of packets that has been transmitted.

plot((n0:nf).^2, 10.*log10(squeeze(results(1,3,:)./results(1,7,:) )./9600)', '-*')
ylabel('Energy per bit [dB]')
xlabel('Number of nodes [n]')
grid

figure()
% In the second column of the matrix, we have the information regarding the
% average packet end-to-end delay.

plot((n0:nf).^2, squeeze(results(1,2,:)), '-*')
ylabel('Average end-to-end delay [s]')
xlabel('Number of nodes [n]')
grid

figure()
% In the sixth column of the matrix, we have the information regarding the
% number of collisions.

plot((n0:nf).^2, squeeze(results(1,6,:)), '-*')
ylabel('Number of collisions')
xlabel('Number of nodes [n]')
grid
```

Bibliography

[1] AUVNetSim Project Site, http://sourceforge.net/projects/auvnetsim/.

[2] Guido van Rossum, "Python Tutorial", http://docs.python.org/tut/.

[3] T.Vignaux, K.Muller, "The SimPy Manual", http://simpy.sourceforge.net/.

[4] Python Project Site, http://www.python.org/download/.

[5] SimPy Project Site, http://simpy.sourceforge.net/archive.htm.

[6] MatPlot Library Project Site, http://sourceforge.net/projects/matplotlib/.

[7] NumPy Project Site, http://numpy.scipy.org/.

[8] TortoiseSVN Site, https://sourceforge.net/projects/tortoisesvn/

[9] AUVNetSim SVN repository, https://auvnetsim.svn.sourceforge.net/svnroot/auvnetsim