AUVNetSim A SIMULATOR FOR UNDERWATER ACOUSTIC NETWORKS

Josep Miquel Jornet Montana

MITSG 08-4

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

NOAA Grant No.: NA060AR4170016

AUVNetSim: A Simulator for Underwater Acoustic Networks

Josep Miquel Jornet Montana

May 14, 2008

AUVNetSim is a simulation library for testing acoustic networking algorithms [1]. It is written in Python [2] and it 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 the advantage of the existing structure.

1 Prerequisites

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

- Python Environment: the python core software [4].
- SimPy Package: a discrete-event simulation system [5].
- MatplotLib: a python plotting library [6].
- Numpy: a package that provides scientific computing functionalities [7].

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

2 AUVNetSim for End Users

The simulator already contains a great variety of parameters and protocols that can be selected. 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 is 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"
    nodes = AUVSim.RunSimulation(config)    # The simulation is launched
    print "Done"

    PlotScenario(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. After that, 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 = 1800.00

# Available Bandwidth (kHz)
BandWidth = 48.00

# Bandwidth efficiency (bps/Hz)
BandwidthBitrateRelation = 1.00
```

```
# Frequency (kHz)
Frequency = 44.00

# Maximum Transmit Power -> Acoustic Intensity (dB re uPa)
TransmitPower = 500.00

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

# Listen Power (dB) -> Battery Consumption (dB)
ListenPowerW = -10.00

# 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 parameteres
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": "FBR", "variation":0, "coneAngle":60.0}

# Nodes: here is where we define individual nodes
# format: AcousticNode(Address, position[, period, destination])
Nodes = [["A", (0,9000,1000), 4*60, "Sink"],["D",(9000,0,1000), 4*60, "Sink"],
["B", (9000,9000,1000), 4*60, "Sink"], ["C", (0,0,1000), 4*60, "Sink"],
["Sink",(4500,4500,1000), None, "A"]]
```

All the possible parameters that can be currently specified are summarized in Table 1. The simulation can be started by just typing from the OS command line:

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

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].

Physical Layer		
Center Frequency	[kHz]	
Bandwidth	[kHz]	
Bandwidth Efficiency	[bit/Hz]	
Transmitting mode max power	[dB]	
Receiving power consumption	[dB]	
Listening power consumption	[dB]	
Listening threshold	[dB]	
Receiving threshold	[dB]	
Power Control	True/False	
Power Levels	name:value[km]	Only if P.Control is used
Medium Access Control		
Protocol	CS-ALOHA, DACAP	
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 attemps		
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 if FBR is used
Cone aperture	[degree]	Only if FBR is used
Retransmission attemps		Only if FBR is used
Nodes		
Name		
Period	[s]	Can be None
Destination	[node]	Can be None
Position or Path	List of points	
Simulation Duration	[s]	

Table 1: AUVNetSim parameters that should be specified in the configuration file

3.1 Simulator Structure

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.

3.1.1 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()

    # Signal all nodes that a message has been transmitted
    AcousticEvent = Sim.SimEvent("AcousticEvent")

    nodes = SetupNodesForSimulation(config_dict, AcousticEvent)

    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.

```
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
```

3.1.2 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.

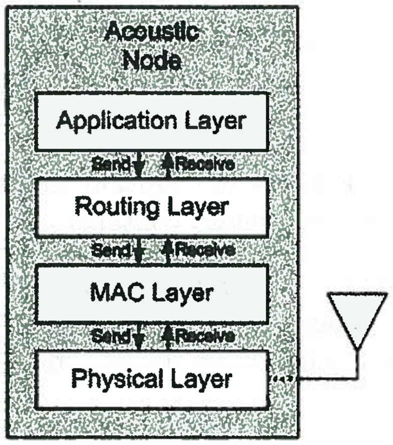

Figure 1: AUVNetSim: node programming structure.

```
class AcousticNode():

    def __init__(self, event, config, label, position, period=None, destination=None):
        self.config = config
        self.name = label        # Node name
        self.random_dest = False # Indicates if the destination is fixed
                                 # or randomly selected

        self.SetupPath(position_or_path) # Position or path of the node
        self.total = nHigh*nWide

        # 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:
            # Schedules the first transmission
            self.SetUpPeriodicTransmission(config["Period"], destination)
```

3.1.3 PhysicalLayer.py

The physical layer of an acoustic node is modeled by a modem and a transducer. The modem operates in half-duplex mode (it can only receive or transmit at a time). When a packet is received from the MAC layer, the modem will automatically change to transmission mode, even if there were packets being received. It is the MAC protocol's duty to check if the channel is idle before transmitting. This information is obtained from the modem. When a packet is being received through the acoustic transducer, the modem is in reception mode. At the end of the reception, a packet can be either properly received and passed on the MAC layer; just overheard (received but without a pre-specified SNR); or discarded because of interference. When a modem detects a collision, it is possible to inform the MAC layer.

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)
```

3.1.4 MAC.py

Different MAC protocols are defined in this file. CS-ALOHA, DACAP and DACAP for FBR 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 functionalities 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])
```

3.1.5 Routing Layer

A programmer willing to include a new routing technique should do it in this file. 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)
```

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

3.1.6 Application Layer

In the application layer, packets may be periodically generated and relayed to the lower layers. 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 self.random_dest and destination==None:
            num = randint(0, self.node.total)
            destination = self.node.prefix+'%03d'%(num,)

            if destination == self.node.name or destination == "S000":
                destination = "Sink"

        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)
```

3.1.7 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.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. Only the nodes that participated in the packets transmissions are shown by name.
2. 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.
3. 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.
4. Routes can then visually be identified.

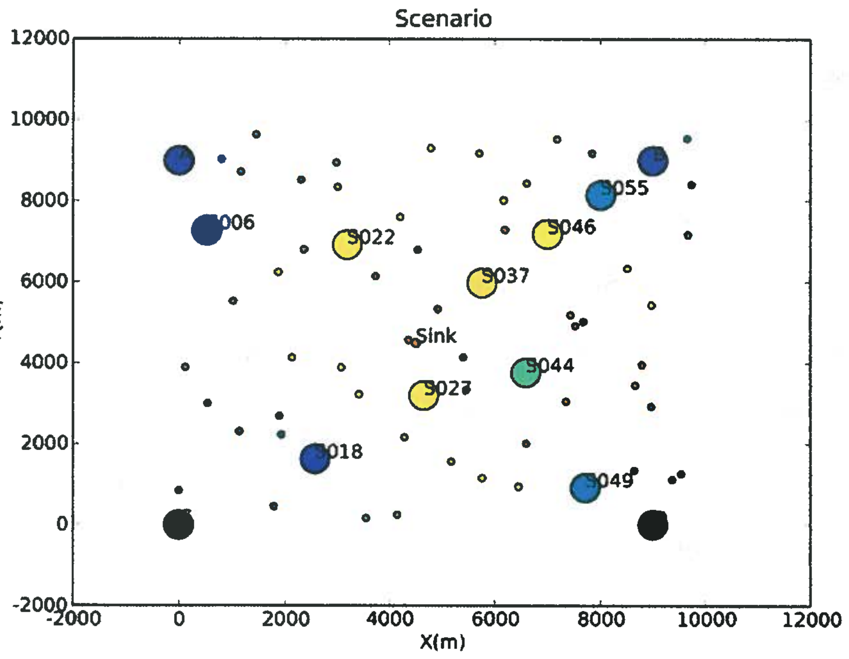

Figure 2: AUVNetSim: scenario for simulation and final node status.

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. Fig.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.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.

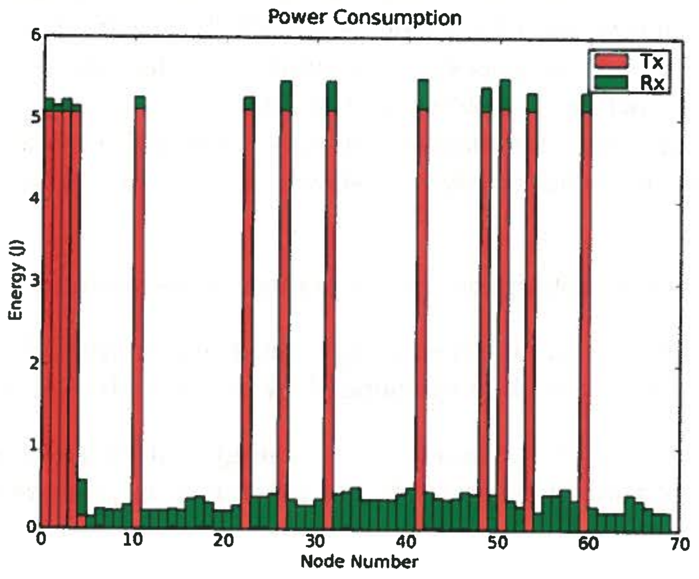

Figure 3: AUVNetSim: energy consumption at each node of the scenario.

References

[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/.

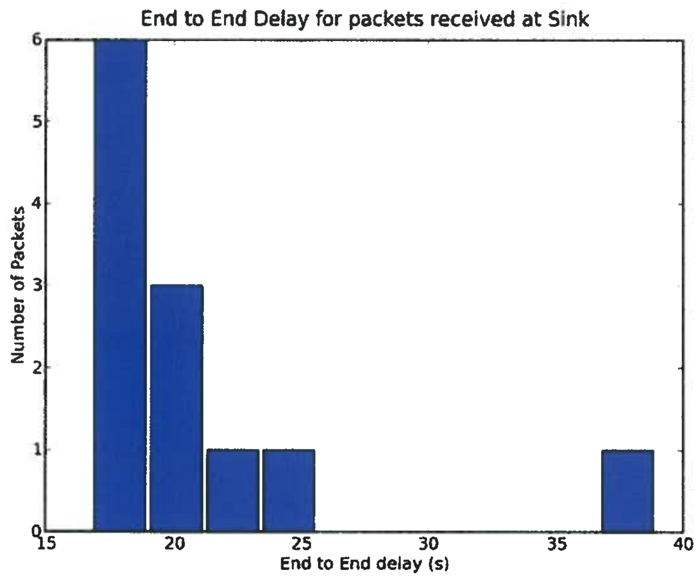

Figure 4: AUVNetSim: histogram of the end-to-end delay of the packets received at the common sink.

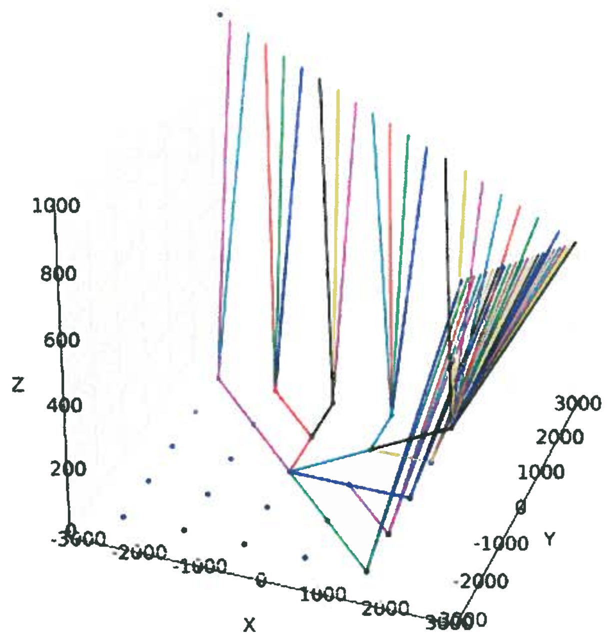

Figure 5: AUVNetSim: 3D scenario containing an AUV and a node-field with 16 relays.