Mid-Sem Report

Design Oriented Project

<u>Wireless Underwater Networks</u>

Birla Institute of Technology and Science,

Pilani (KK Birla Goa Campus)

Authors:
Umang Gupta(2017A8PS0510G)
Krishna Pranay Reddy Chennareddy(2018AAPS0302G)
Amrit M (2018AAPS0700G)
Krishi Ajaykumar Agrawal(2019AAPS0235G)

Guided by: Dr. Sarang C. Dhongdi.

Contents

Acknowledgements

We'd like to express our sincere gratitude to *Dr. Sarang Dhongdi* Sir for his constant support and guidance throughout this project.

Introduction

In the past 30 years we have observed major growth in underwater communications because of its varying applications such as marine research, oceanography, marine commercial operations, the offshore oil industry and defense. Continued research over the years has resulted in improved performance and robustness as compared to the initial communication systems. High-speed communication in the underwater acoustic channel has been challenging because of limited bandwidth, extended multipath, refractive properties of the medium, severe fading, rapid time variation and large Doppler Shifts. In the initial years, rapid progress was made in deepwater communication, but the shallow water channel was considered difficult.

Through this project we aim to explore various mobile underwater network models that can be set up and used for communication. For our experiments and simulations we will be using UnetStack. For the first half of the process our aim was to explore different models available within UnetStack and to experiment with the range and MAC protocols available within UnetStack. As our project develops we'll look into implementing the various protocols available for all layers within the protocol stack and set up an optimal underwater network.

We will be using the UnetStack framework for our simulations.The Unet project strives to develop technologies that allow us to build communication networks that extend underwater, be it via acoustic, optical, or even wired links. Some nodes in such networks may be above water, while others are underwater. In this handbook, we explore how to build such networks using UnetStack 3, an agent-based network technology that was developed in the Unet project.

Experimentation with Routing

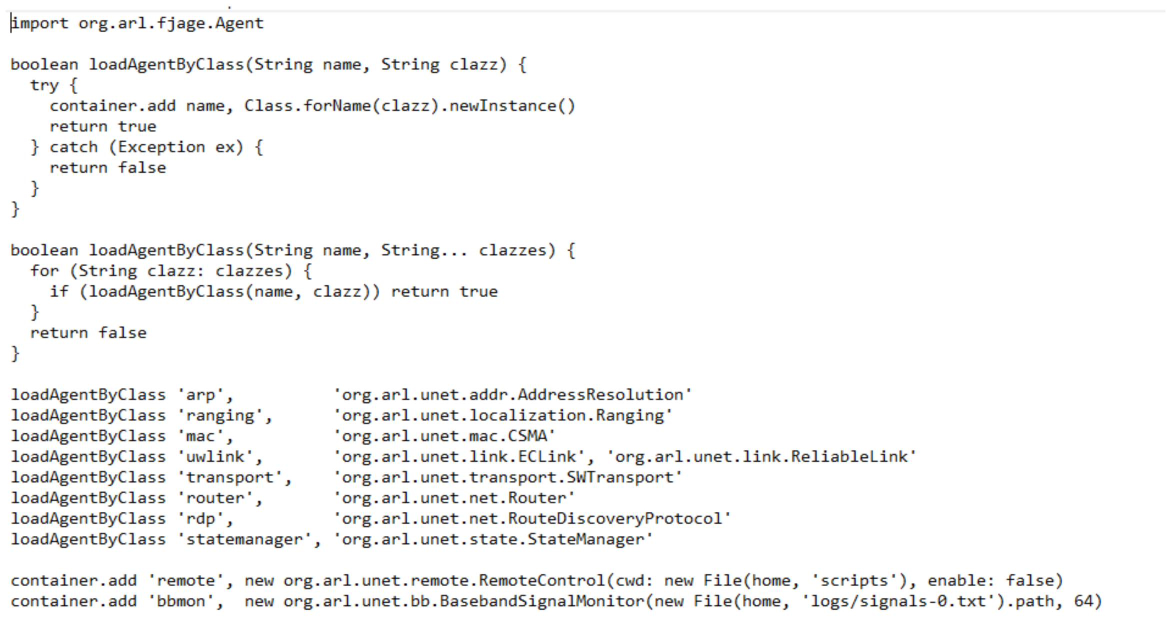
For the first experiment our aim was to set up a three node underwater network and experiment with different routing procedures. Through this experiment we also aimed to develop a better understanding of the ranging and power limitation of the nodes at different depths provided within the module.

Procedure:

1.) We used the RealTimePlatform class to create a real time simulator for the nodes.
   - a.) RealTimePlatform runs the agents in its containers in real-time. The notion of time in this platform is that of elapsed time in the real world, and hence this is a suitable platform for most applications.
   - b.) Inherited methods from class Platform: addContainer, getBuildVersion, getContainers, getHostname, getNetworkInterface, getPort, isIdle, isRunning, setHostname, setNetworkInterface, setNetworkInterface, setPort, shutdown, start. There are the very primary classes that provide the basic functionality for our simulations

2.) We used the TCP/IP networking model.

3.) We initiated the simulation and we used the stack originally provided for creating the nodes.



```
import org.arl.fjage.Agent

boolean loadAgentByClass(String name, String clazz) {
  try {
    container.add name, Class.forName(clazz).newInstance()
    return true
  } catch (Exception ex) {
    return false
  }
}

boolean loadAgentByClass(String name, String... clazzes) {
  for (String clazz: clazzes) {
    if (loadAgentByClass(name, clazz)) return true
  }
  return false
}

loadAgentByClass 'arp',          'org.arl.unet.addr.AddressResolution'
loadAgentByClass 'ranging',      'org.arl.unet.localization.Ranging'
loadAgentByClass 'mac',          'org.arl.unet.mac.CSMA'
loadAgentByClass 'uwlink',       'org.arl.unet.link.ECLink', 'org.arl.unet.link.ReliableLink'
loadAgentByClass 'transport',    'org.arl.unet.transport.SWTransport'
loadAgentByClass 'router',       'org.arl.unet.net.Router'
loadAgentByClass 'rdp',          'org.arl.unet.net.RouteDiscoveryProtocol'
loadAgentByClass 'statemanager', 'org.arl.unet.state.StateManager'

container.add 'remote', new org.arl.unet.remote.RemoteControl(cwd: new File(home, 'scripts'), enable: false)
container.add 'bbmon',  new org.arl.unet.bb.BasebandSignalMonitor(new File(home, 'logs/signals-0.txt').path, 64)
```

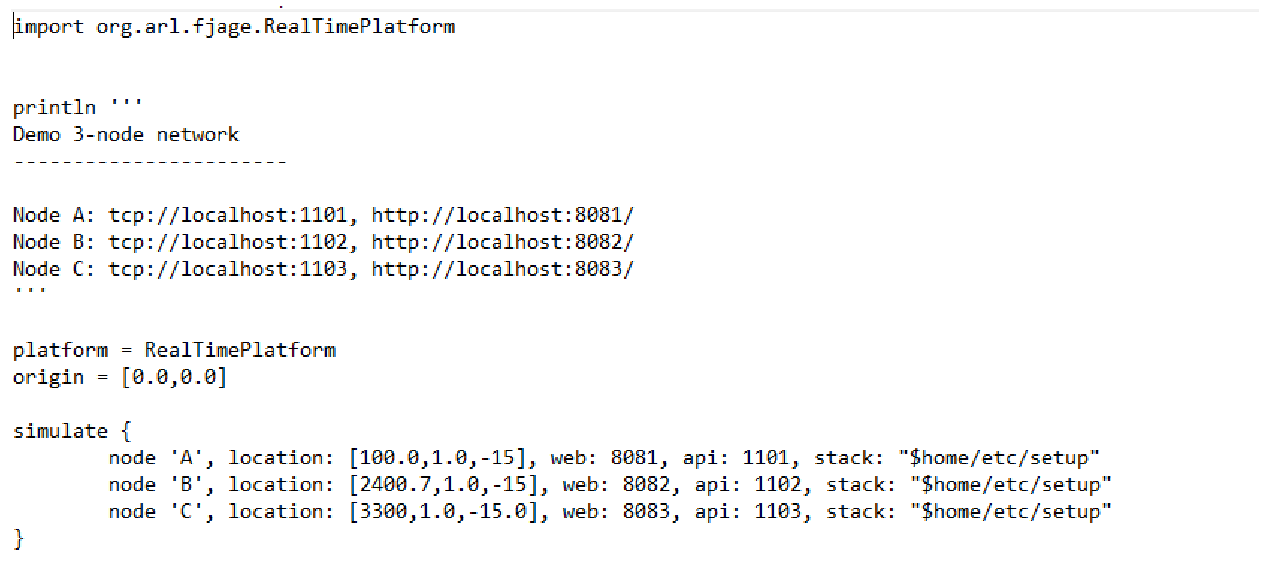
a.) import the class Agent because it is the Base class to be extended by all agents. An agent must be added to a container in order to run. An agent usually overrides the init() and shutdown() methods to provide the appropriate behavior.

b.) Then we load all the agents like ones for transport, ranging, routing etc.

c.) We used CSMA for MAC.

4.) Finally we set different distances for our nodes to check at which range the signals start breaking. The distances between the nodes was decided upon by trial and error checking for any packet losses at different distances.

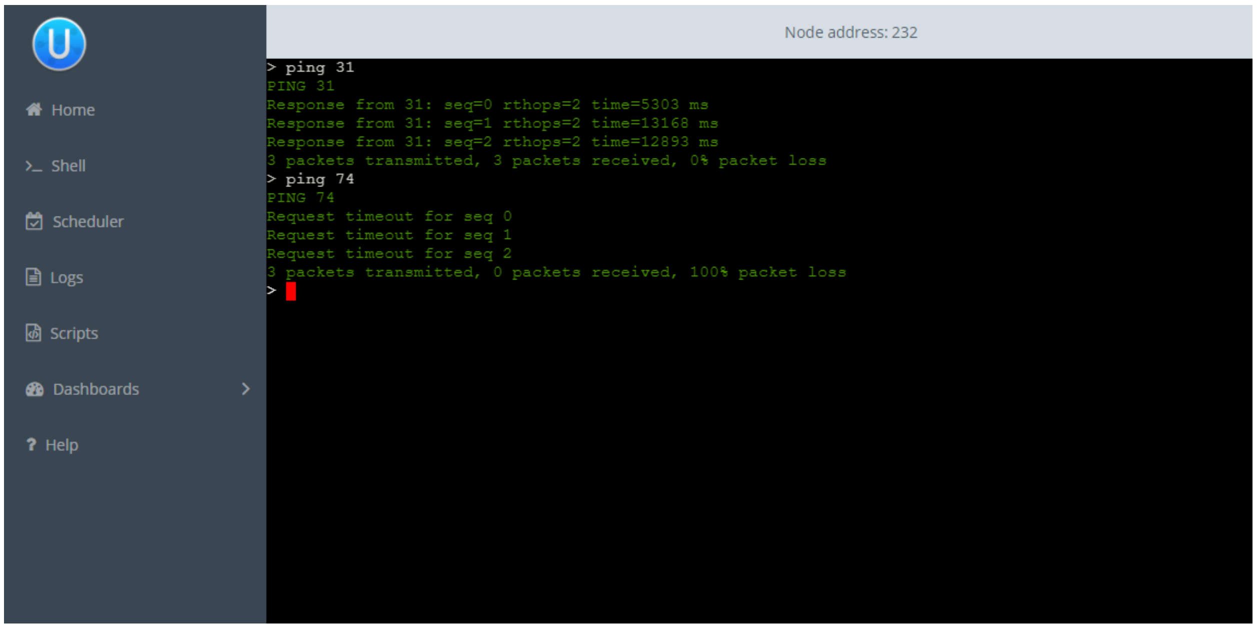
Here’s the groovy setup for our demonstration.



```
import org.arl.fjage.RealTimePlatform


println '''
Demo 3-node network
-----------------------

Node A: tcp://localhost:1101, http://localhost:8081/
Node B: tcp://localhost:1102, http://localhost:8082/
Node C: tcp://localhost:1103, http://localhost:8083/
'''

platform = RealTimePlatform
origin = [0.0,0.0]

simulate {
        node 'A', location: [100.0,1.0,-15], web: 8081, api: 1101, stack: "$home/etc/setup"
        node 'B', location: [2400.7,1.0,-15], web: 8082, api: 1102, stack: "$home/etc/setup"
        node 'C', location: [3300,1.0,-15.0], web: 8083, api: 1103, stack: "$home/etc/setup"
}
```

Results

1.) We check connectivity to the other nodes using the ‘`ping`’ command. We see that Node ‘B’ is reachable and there is **0% loss** and Node ‘C’ is unreachable. Node ‘B’ is at a distance 2.3 km from the Node ‘A’ and the Node ’C’ is at a distance of 4.1 km from Node ‘A’.

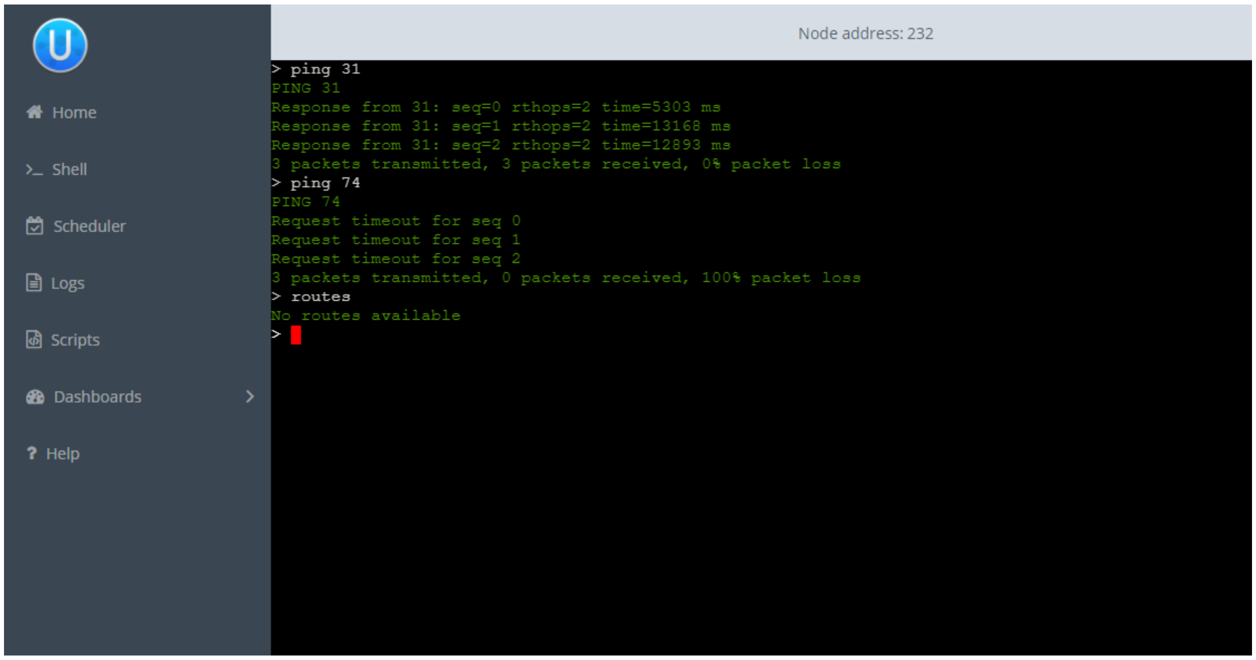


Node address: 232

```
> ping 31
PING 31
Response from 31: seq=0 rthops=2 time=5303 ms
Response from 31: seq=1 rthops=2 time=13168 ms
Response from 31: seq=2 rthops=2 time=12893 ms
3 packets transmitted, 3 packets received, 0% packet loss
> ping 74
PING 74
Request timeout for seq 0
Request timeout for seq 1
Request timeout for seq 2
3 packets transmitted, 0 packets received, 100% packet loss
> 
```

2.) Now we'll check if there are any routes from Node 'A' to Node 'C'.

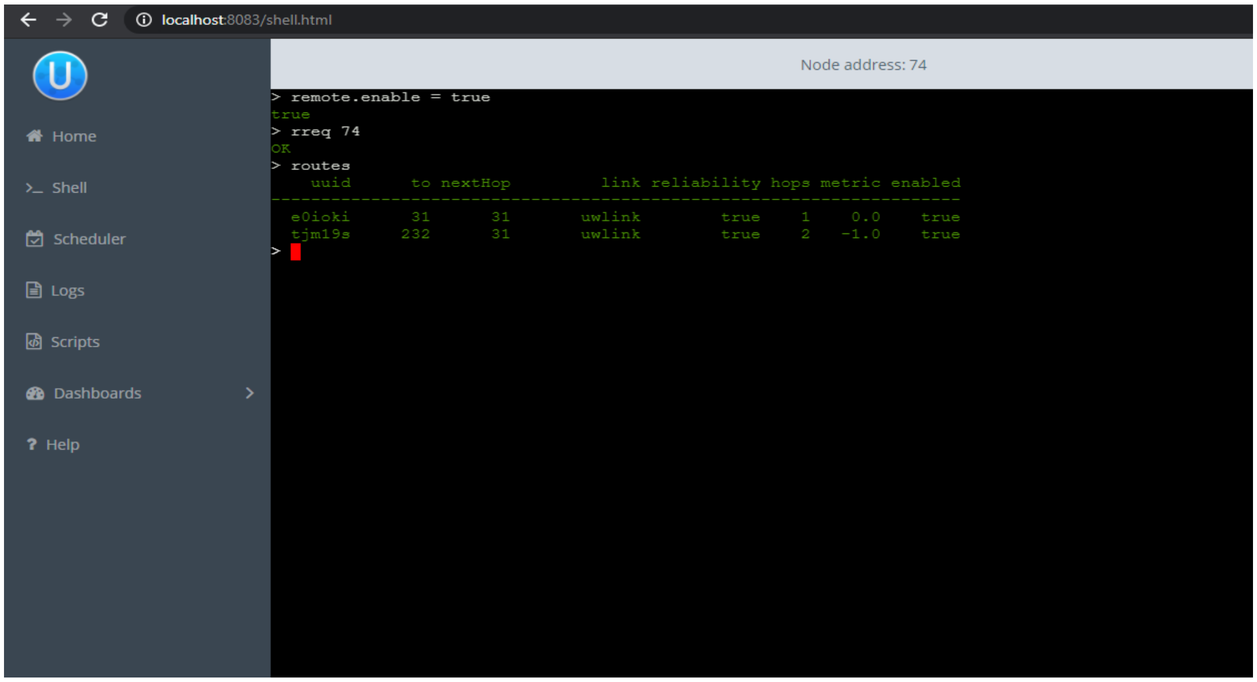
3.) Now to add routes we'll enable remote access to all the nodes and use the '`rreq`'



```
> ping 31
PING 31
Response from 31: seq=0 rthops=2 time=5303 ms
Response from 31: seq=1 rthops=2 time=13168 ms
Response from 31: seq=2 rthops=2 time=12893 ms
3 packets transmitted, 3 packets received, 0% packet loss
> ping 74
PING 74
Request timeout for seq 0
Request timeout for seq 1
Request timeout for seq 2
3 packets transmitted, 0 packets received, 100% packet loss
> routes
No routes available
>
```

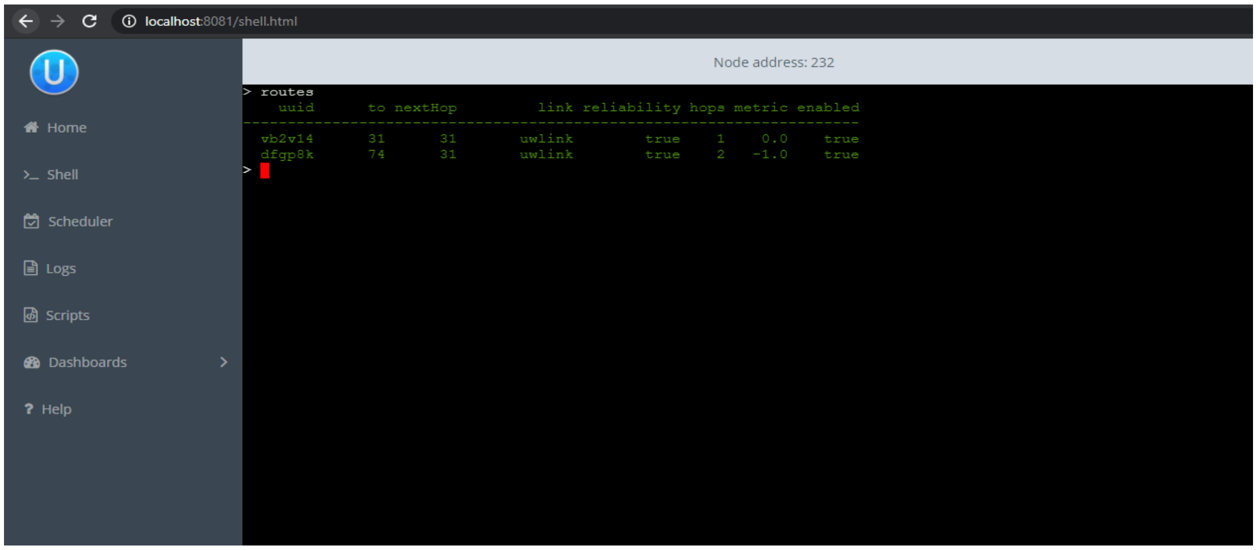
command mentioned in the Routing Protocols.

4.) This adds routes from A via B to C and also at C it adds a route from C to A via B.



```
> remote.enable = true
true
> rreq 74
OK
> routes
   uuid      to nextHop         link reliability hops metric enabled
--------------------------------------------------------------------
  e0ioki      31      31       uwlink         true    1    0.0    true
  tjm19s     232      31       uwlink         true    2   -1.0    true
>
```

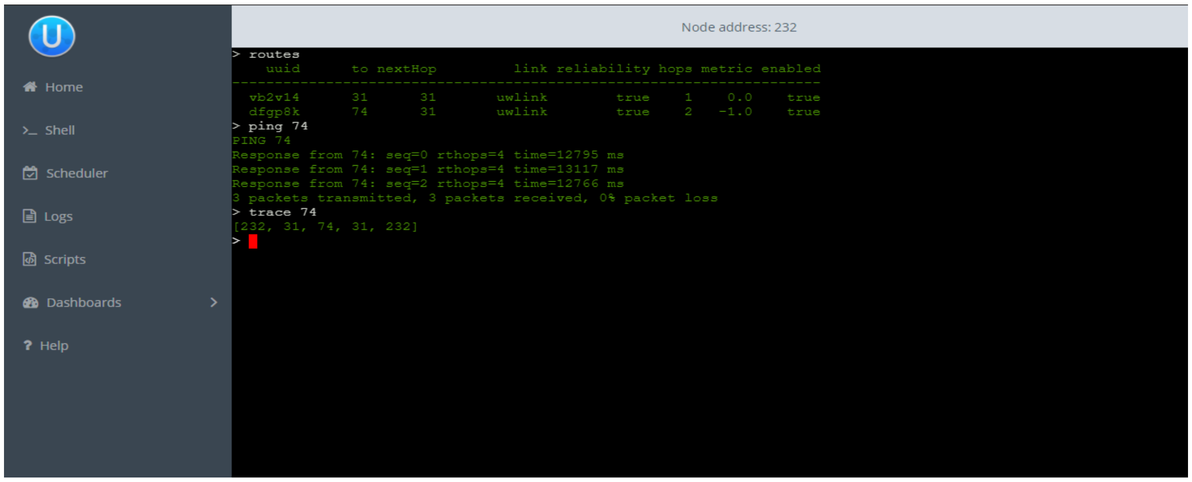
5.) We use routes to check the routing process and observe the routes have been added.



```
> routes
   uuid      to nextHop         link reliability hops metric enabled
--------------------------------------------------------------------
  vb2v14     31      31       uwlink        true    1    0.0    true
  dfgp8k     74      31       uwlink        true    2   -1.0    true
>
```

6.) Now we'll check the connectivity again using 'ping' and then check the routing using '`trace`'.

7.) Trace command shows that there is a route to 'C' and it is through node 'B' and there is no data loss.



```
> routes
   uuid      to nextHop         link reliability hops metric enabled
--------------------------------------------------------------------
  vb2v14     31      31       uwlink        true    1    0.0    true
  dfgp8k     74      31       uwlink        true    2   -1.0    true
> ping 74
PING 74
Response from 74: seq=0 rthops=4 time=12795 ms
Response from 74: seq=1 rthops=4 time=13117 ms
Response from 74: seq=2 rthops=4 time=12766 ms
3 packets transmitted, 3 packets received, 0% packet loss
> trace 74
[232, 31, 74, 31, 232]
>
```

8.) We can also add these routes manually using '`addroute`'.

Observations

1.) We can connect nodes for a range of 2.5km such that no data loss takes place with a power level of -10 dB re 1 μPa^2/Hz operating at a depth of 15m.

2.) For the same depth, with the powel level of 0 dB re 1 μPa^2/Hz, the maximum range observed was about 4km.

Experimentation with Different MAC procedures

For the second experiment we explored the various models of MAC available in UnetStack.

Why MAC protocols ?

Key differences between terrestrial radio wireless networking and underwater acoustic networking are the large propagation delay of sound, extremely low point-to-point data rates and high raw BER. Thus it is often proposed that media access control(MAC) protocols for underwater networks are developed ground up and not directly adopt existing terrestrial protocols. To put us in a position to understand the existing MAC protocols within UNET and eventually develop our own protocols.

MAC protocols use PDUs for channel reservation and piggyback of the client data in the PDU. Such support is available and it is advertised using a non-zero ***reservationPayloadSize*** parameter. A ***ReservationReq*** should provide the payload data to be sent to a peer node (as part of RTS or equivalent PDU) to whom the reservation is made. If that node wishes to send payload data back (as part of CTS or equivalent PDU), it may send a ***ReservationAcceptReq*** in response to a ***ReservationStatusNtf*** to provide its payload data. Protocol data units (PDUs) are protocol-specific datagrams exchanged by nodes in a network. MAC protocols that use a handshake often use three basic types of PDUs — request to send (RTS), clear to send (CTS) and acknowledgement (ACK).

Parameters

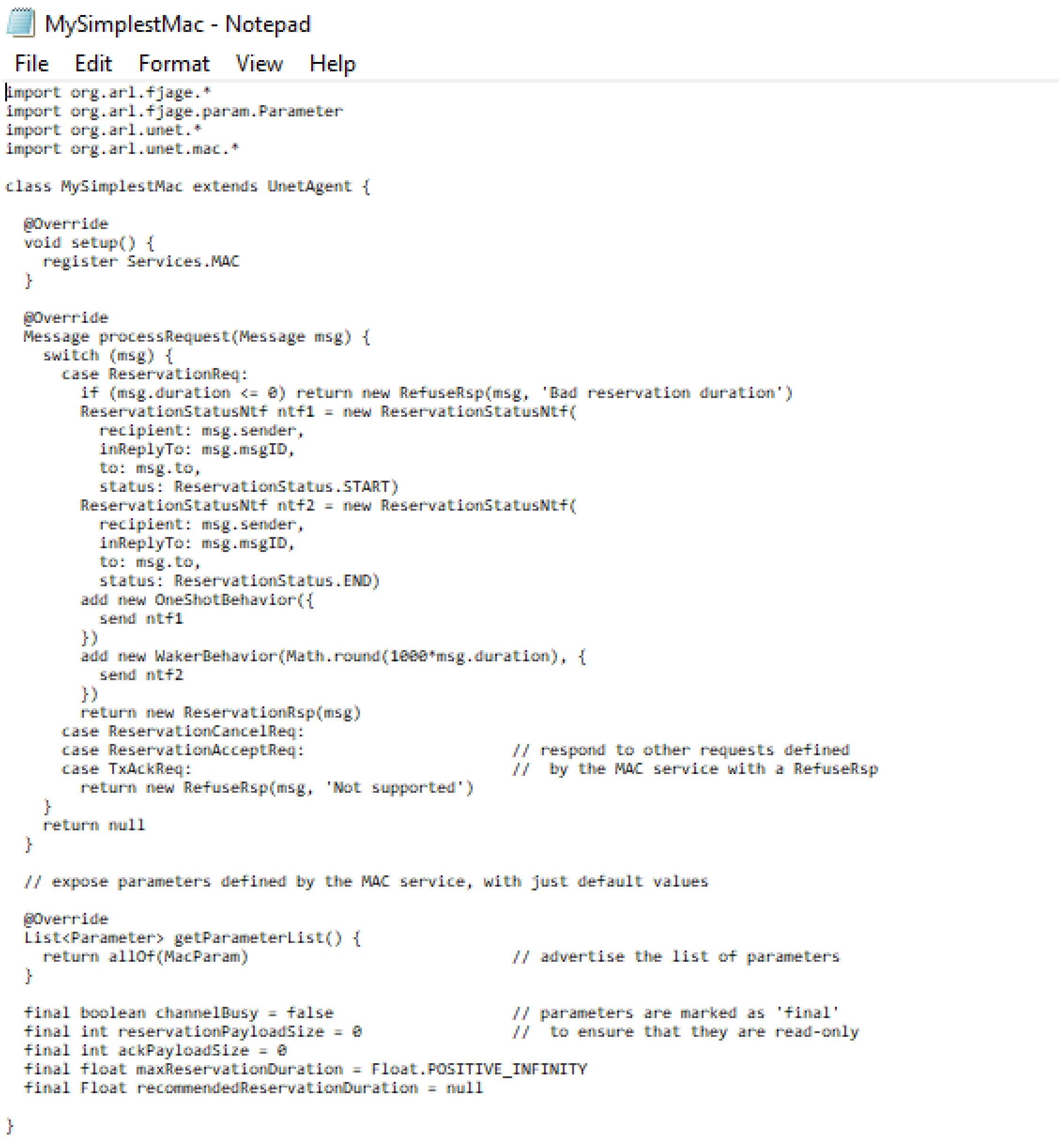
Agents offering the MAC service support the following parameters:

- channelBusy — true if channel is busy, false otherwise
- reservationPayloadSize — maximum size of payload (bytes) that can be piggybacked in reservation PDU
- ackPayloadSize — maximum size of acknowledgement (bytes) that can be included in an ACK PDU
- maxReservationDuration — maximum duration of reservation in seconds
- recommendedReservationDuration — recommended duration of reservation in seconds (null, if unspecified)

MAC protocol stack

There are three different MAC protocols that are provided within UnetStack. First is a Multi-Access Implementation without any collision detection for low traffic setups. The logic for this setup is whenever a request is made the MAC agent checks if the node is not already processing a request, it will setup the OneShotBehavior module that starts the transmission process and after completion starts the WakerBehaviour module that ends the transmission.

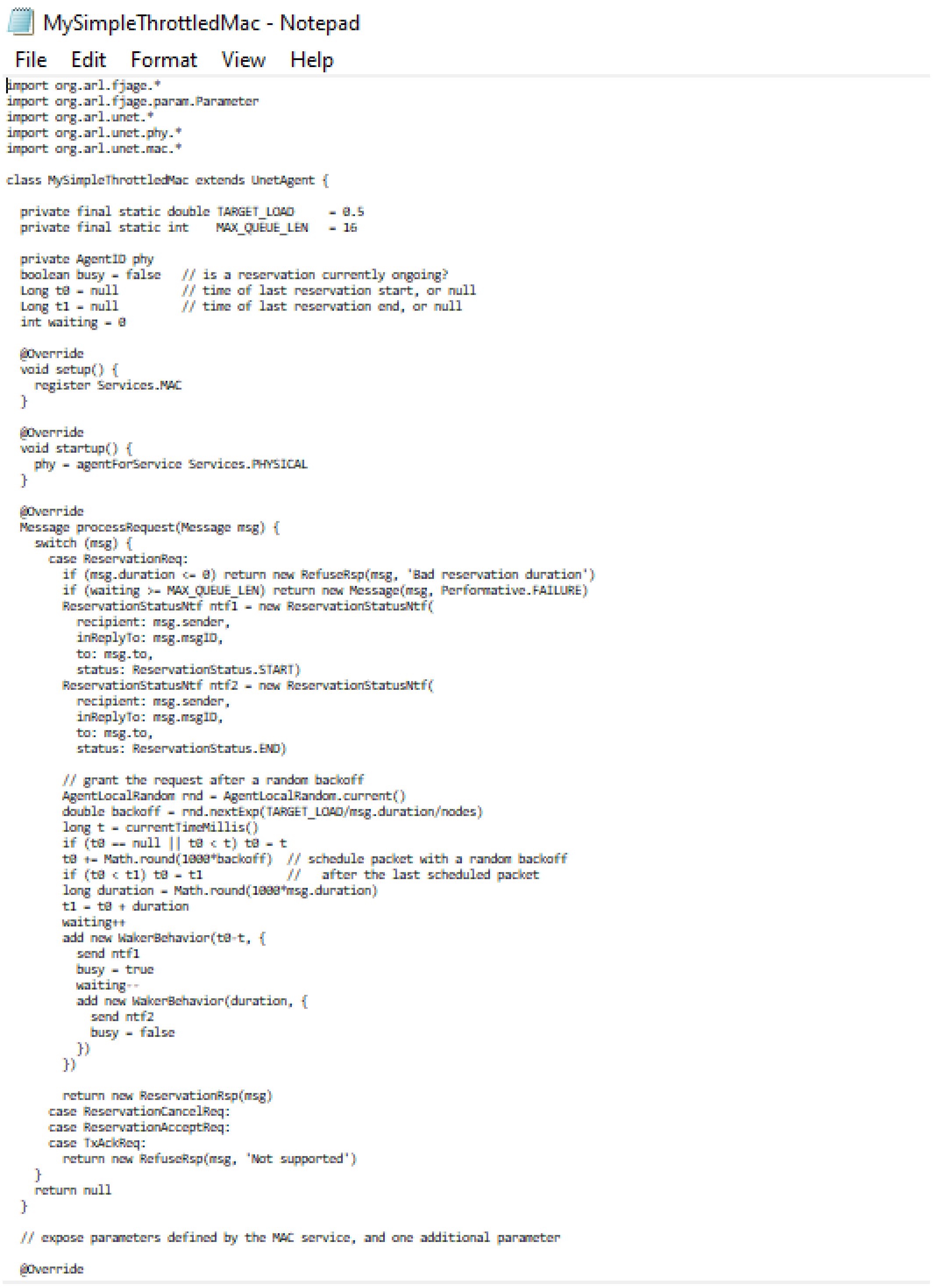
The code for this setup is:



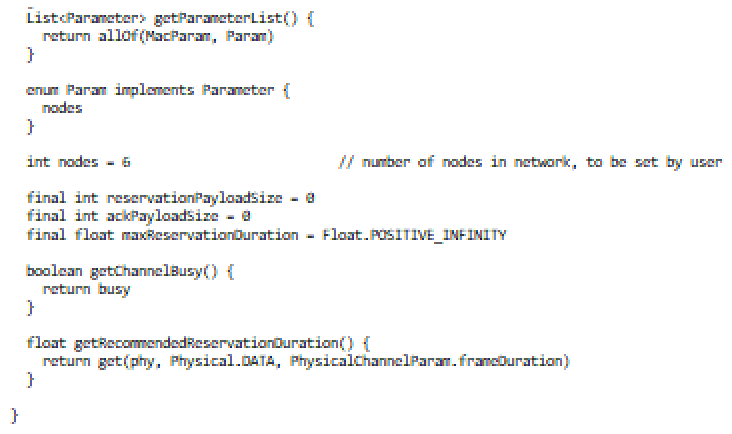
```
import org.arl.fjage.*
import org.arl.fjage.param.Parameter
import org.arl.unet.*
import org.arl.unet.mac.*

class MySimplestMac extends UnetAgent {

  @Override
  void setup() {
    register Services.MAC
  }

  @Override
  Message processRequest(Message msg) {
    switch (msg) {
      case ReservationReq:
        if (msg.duration <= 0) return new RefuseRsp(msg, 'Bad reservation duration')
        ReservationStatusNtf ntf1 = new ReservationStatusNtf(
          recipient: msg.sender,
          inReplyTo: msg.msgID,
          to: msg.to,
          status: ReservationStatus.START)
        ReservationStatusNtf ntf2 = new ReservationStatusNtf(
          recipient: msg.sender,
          inReplyTo: msg.msgID,
          to: msg.to,
          status: ReservationStatus.END)
        add new OneShotBehavior({
          send ntf1
        })
        add new WakerBehavior(Math.round(1000*msg.duration), {
          send ntf2
        })
        return new ReservationRsp(msg)
      case ReservationCancelReq:
      case ReservationAcceptReq:                       // respond to other requests defined
      case TxAckReq:                                   //  by the MAC service with a RefuseRsp
        return new RefuseRsp(msg, 'Not supported')
    }
    return null
  }

  // expose parameters defined by the MAC service, with just default values

  @Override
  List<Parameter> getParameterList() {
    return allOf(MacParam)                             // advertise the list of parameters
  }

  final boolean channelBusy = false                    // parameters are marked as 'final'
  final int reservationPayloadSize = 0                 //  to ensure that they are read-only
  final int ackPayloadSize = 0
  final float maxReservationDuration = Float.POSITIVE_INFINITY
  final Float recommendedReservationDuration = null

}
```

While the above simple MAC would work well when the traffic offered to it is random, it will perform poorly if the network is fully loaded. All nodes would constantly try to access the channel, collide and the throughput would plummet. To address this concern, one may add an exponentially distributed random backoff (Poisson arrival to match the assumption of Aloha) for every request, to introduce randomness. The backoff could be chosen to offer a normalized network load of approximately 0.5, since this generates the highest throughput for Aloha.

Code for this implementation:



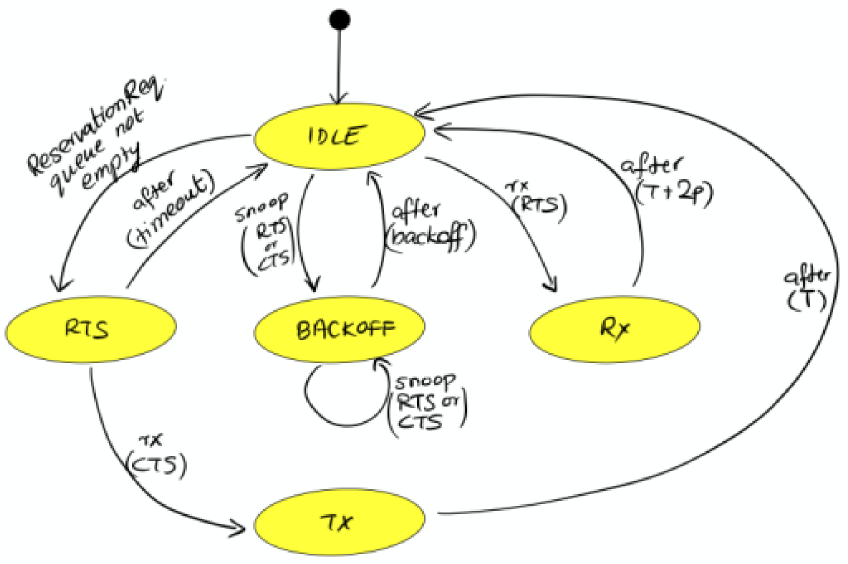
```
import org.arl.fjage.*
import org.arl.fjage.param.Parameter
import org.arl.unet.*
import org.arl.unet.phy.*
import org.arl.unet.mac.*

class MySimpleThrottledMac extends UnetAgent {

  private final static double TARGET_LOAD     = 0.5
  private final static int    MAX_QUEUE_LEN   = 16

  private AgentID phy
  boolean busy = false   // is a reservation currently ongoing?
  Long t0 = null         // time of last reservation start, or null
  Long t1 = null         // time of last reservation end, or null
  int waiting = 0

  @Override
  void setup() {
    register Services.MAC
  }

  @Override
  void startup() {
    phy = agentForService Services.PHYSICAL
  }

  @Override
  Message processRequest(Message msg) {
    switch (msg) {
      case ReservationReq:
        if (msg.duration <= 0) return new RefuseRsp(msg, 'Bad reservation duration')
        if (waiting >= MAX_QUEUE_LEN) return new Message(msg, Performative.FAILURE)
        ReservationStatusNtf ntf1 = new ReservationStatusNtf(
          recipient: msg.sender,
          inReplyTo: msg.msgID,
          to: msg.to,
          status: ReservationStatus.START)
        ReservationStatusNtf ntf2 = new ReservationStatusNtf(
          recipient: msg.sender,
          inReplyTo: msg.msgID,
          to: msg.to,
          status: ReservationStatus.END)

        // grant the request after a random backoff
        AgentLocalRandom rnd = AgentLocalRandom.current()
        double backoff = rnd.nextExp(TARGET_LOAD/msg.duration/nodes)
        long t = currentTimeMillis()
        if (t0 == null || t0 < t) t0 = t
        t0 += Math.round(1000*backoff)  // schedule packet with a random backoff
        if (t0 < t1) t0 = t1            //   after the last scheduled packet
        long duration = Math.round(1000*msg.duration)
        t1 = t0 + duration
        waiting++
        add new WakerBehavior(t0-t, {
          send ntf1
          busy = true
          waiting--
          add new WakerBehavior(duration, {
            send ntf2
            busy = false
          })
        })

        return new ReservationRsp(msg)
      case ReservationCancelReq:
      case ReservationAcceptReq:
      case TxAckReq:
        return new RefuseRsp(msg, 'Not supported')
    }
    return null
  }

  // expose parameters defined by the MAC service, and one additional parameter

  @Override
```



```
  List<Parameter> getParameterList() {
    return allOf(MacParam, Param)
  }

  enum Param implements Parameter {
    nodes
  }

  int nodes = 6                       // number of nodes in network, to be set by user

  final int reservationPayloadSize = 0
  final int ackPayloadSize = 0
  final float maxReservationDuration = Float.POSITIVE_INFINITY

  boolean getChannelBusy() {
    return busy
  }

  float getRecommendedReservationDuration() {
    return get(phy, Physical.DATA, PhysicalChannelParam.frameDuration)
  }

}
```

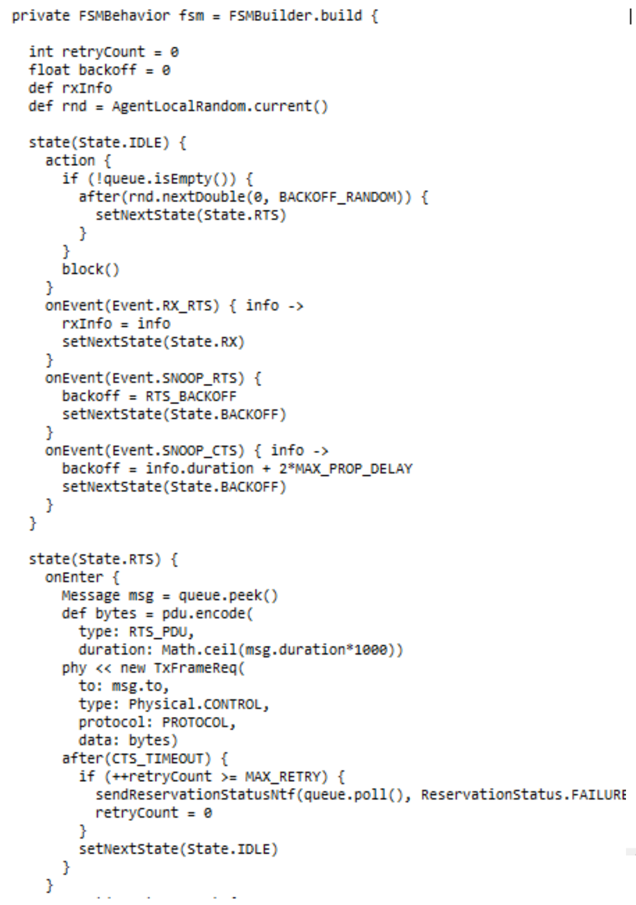
Random numbers are generated using the AgentLocalRandom utility. This utility ensures repeatable results during discrete event simulation, aiding with debugging, and so is the preferred way of generating random variates. The nextExp() function generates an exponentially distributed random number with a specified rate parameter. The rate parameter is computed such that the average backoff introduced helps to achieve the specified target load. Now, we no longer send the START notification immediately instead we schedule it after a backoff, and then schedule the END notification after the reservation duration from the START.

For other MAC protocols like MACA and FAMA we'll require a protocol that allows handshake or some sort of acknowledgement. As per a paper by Mandar Chitre and Shiraz Shahabudeen on Underwater Acoustic Communications, MACA protocols are extremely efficient in simulation for communication. Thus, these protocols and their implementation are important while exploring different underwater network models. To implement these models UnetStack uses Finite State Machines. FSMs are very commonly used in network protocol development and for the Simple Handshake Model the FSM used is given below:

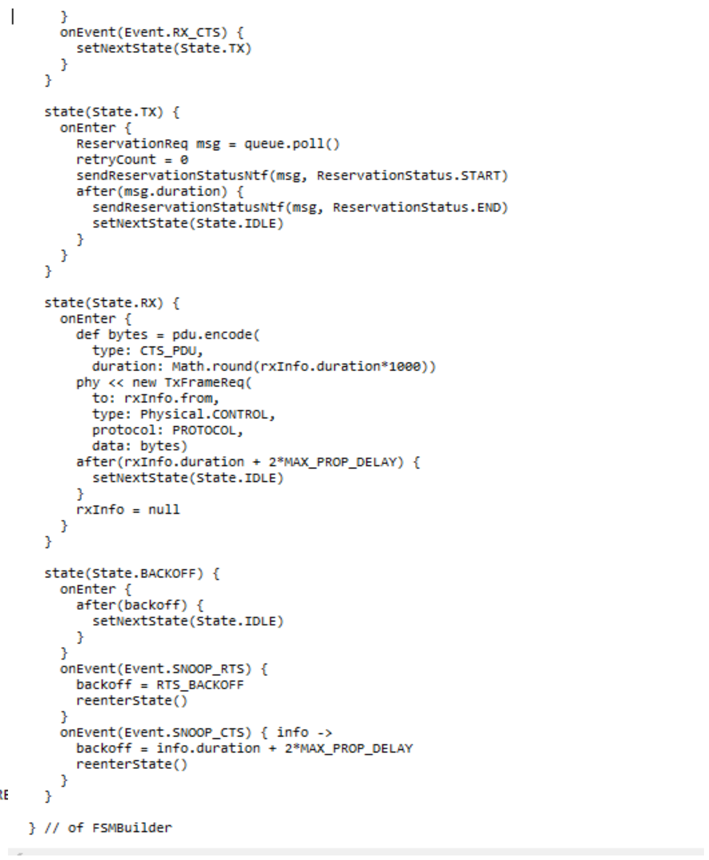


When the channel is free, the agent is in an IDLE state. If the agent receives a ReservationReq, it switches to the RTS state and sends an RTS PDU to the intended destination node. If it receives a CTS PDU back, then it switches to a TX state and urges the client to transmit data via a ReservationStatusNtf with a START status. After the reservation period is over, the agent switches back to the IDLE state. If no CTS PDU is received in the RTS state for a while, the agent times out and returns to the IDLE state after informing the client of a reservation FAILURE. If the agent receives an RTS PDU in the IDLE state, it switches to the RX state and responds with a CTS PDU. The node initiating the handshake may then transmit data for the reservation duration. After the duration (plus some allowance for 2-way propagation delay), the agent switches back to the IDLE state. If the agent overhears (aka snoops) RTS or CTS PDUs destined for other nodes, it switches to a BACKOFF state for a while. During the state, it does not initiate or respond to RTS PDUs. After the backoff period, it switches back to the IDLE state.

To build these FSM UnetStack provides FSMbuildermodule and the code for our MAC handshake protocol FSM is given here:



```
private FSMBehavior fsm = FSMBuilder.build {

  int retryCount = 0
  float backoff = 0
  def rxInfo
  def rnd = AgentLocalRandom.current()

  state(State.IDLE) {
    action {
      if (!queue.isEmpty()) {
        after(rnd.nextDouble(0, BACKOFF_RANDOM)) {
          setNextState(State.RTS)
        }
      }
      block()
    }
    onEvent(Event.RX_RTS) { info ->
      rxInfo = info
      setNextState(State.RX)
    }
    onEvent(Event.SNOOP_RTS) {
      backoff = RTS_BACKOFF
      setNextState(State.BACKOFF)
    }
    onEvent(Event.SNOOP_CTS) { info ->
      backoff = info.duration + 2*MAX_PROP_DELAY
      setNextState(State.BACKOFF)
    }
  }

  state(State.RTS) {
    onEnter {
      Message msg = queue.peek()
      def bytes = pdu.encode(
        type: RTS_PDU,
        duration: Math.ceil(msg.duration*1000))
      phy << new TxFrameReq(
        to: msg.to,
        type: Physical.CONTROL,
        protocol: PROTOCOL,
        data: bytes)
      after(CTS_TIMEOUT) {
        if (++retryCount >= MAX_RETRY) {
          sendReservationStatusNtf(queue.poll(), ReservationStatus.FAILURE
          retryCount = 0
        }
        setNextState(State.IDLE)
      }
    }
```



```
    }
    onEvent(Event.RX_CTS) {
      setNextState(State.TX)
    }
  }

  state(State.TX) {
    onEnter {
      ReservationReq msg = queue.poll()
      retryCount = 0
      sendReservationStatusNtf(msg, ReservationStatus.START)
      after(msg.duration) {
        sendReservationStatusNtf(msg, ReservationStatus.END)
        setNextState(State.IDLE)
      }
    }
  }

  state(State.RX) {
    onEnter {
      def bytes = pdu.encode(
        type: CTS_PDU,
        duration: Math.round(rxInfo.duration*1000))
      phy << new TxFrameReq(
        to: rxInfo.from,
        type: Physical.CONTROL,
        protocol: PROTOCOL,
        data: bytes)
      after(rxInfo.duration + 2*MAX_PROP_DELAY) {
        setNextState(State.IDLE)
      }
      rxInfo = null
    }
  }

  state(State.BACKOFF) {
    onEnter {
      after(backoff) {
        setNextState(State.IDLE)
      }
    }
    onEvent(Event.SNOOP_RTS) {
      backoff = RTS_BACKOFF
      reenterState()
    }
    onEvent(Event.SNOOP_CTS) { info ->
      backoff = info.duration + 2*MAX_PROP_DELAY
      reenterState()
    }
  }

} // of FSMBuilder
```

The remaining of the MAC protocol implementation utilizes the OneShotBehavior and other modules similar to the previous MAC implementations.

Conclusion

Through the first half of this project we've explored different procedures and protocols provided within UnetStack for simulating underwater communication networks. We also explored the different libraries provided in UnetStack which implement the physical layer and routing procedures initially. Later we also looked into the Mac Protocols within unetstack and started to research the available protocols ,understanding their implementation and analyzing their simulation results.

Progress till Now:

- Successfully compiled and simulated 2-node, 3-node networks on local systems using UnetStack
- Explored Routing Options , Different Ranges of Communication and Different Depth level for Individual nodes and analyzed their behavior
- Explored remote shell accessing of one node from another using `rsh`
- Established Multihop communication network in UnetStack using the Node shell.
- Explored in-built auto routing options such as rreq
- Explored UnetSocket API to establish communication networks for sending and receiving strings of data among nodes using local python/groovy scripts
- Started working on scripting the 3-Node-work setup to automate the routing and setting up an extensible interface for further use.(in progress)
- Looked into different MAC protocols available within the UnetStack framework and understood how they are implemented because these could allow us to implement different protocols like MACA, CDMA, etc.

Notes on Future Work:

- Successfully compile the *script* for the 3-Node-Network setup with proper distances, routes and analyze the results.
- Research different network protocols available for underwater communications and prepare a protocol stack stating options for each layer.
- Generate Trace files for these protocols to explore their efficiency and decide which combinations provide the best results.
- Simulate a UnetStack network implementing this protocol stack.