— DESERT Underwater v2.0.0 —
Description of two representative samples

1 Introduction

In this document we describe in details four tcl samples

- `test_uwcsmaaloha.tcl` that well represents a general structure of a tcl script to test various protocols;
- `S2C_uwpolling_sample.tcl` and `S2C_EvoLogics_uwpolling_sample.sh` scripts that make it possible to test protocols using S2C EvoLogics modems.
- `test_uwoptical_prop.tcl` that explains how to simulate an underwater optical network with inclusion of ambient light noise.
- `test_uwrov.tcl` that explains how to simulate a ROV moving toward the way-points sent by a wireless remote control.

2 Description of `test_uwcsmaaloha.tcl`

We start with `test_uwcsmaaloha.tcl` and explain the most important section of the script.

2.1 Loading the dynamic libraries

```
load libMiracle.so
load libMiracleBasicMovement.so
load libmphy.so
load libmmac.so
load libUwmStd.so
load libuwip.so
load libuwstaticrouting.so
load libuwmll.so
load libuwudp.so
load libuwcbr.so
load libuwcsmaaloha.so
```

In this part we load all the libraries of the modules that we are going to use in the simulation. The order in which the libraries are loaded is important, as all dependencies must be loaded first. Please refer to the doxygen documentation to see all the dependencies of the DESERT modules. In order to find the name of the library of the module you want to use, you can either refer to a sample where the same module is used, or take a look to the `Makefile.am` associated with the module you want to use: in particular, you can refer to the name of the libraries assigned to the variable `lib_LTLIBRARIES`.

2.2 Setting up the simulator

```
set ns [new Simulator]
$ns use-Miracle
```

With the two lines above, we set up the simulator and we tell it to use Miracle (Miracle is needed by DESERT).

2.3 Initialization of the general variables and module configuration

```
set opt(start_clock) [clock seconds]
set opt(nn)          4.0
set opt(pktsize)     125
set opt(starttime)   1
set opt(stoptime)    100000
set opt(txduration)  [expr $opt(stoptime) - $opt(starttime)]
set opt(txpower)     180.0
set opt(maxinterval_) 20.0
set opt(freq)        25000.0
set opt(bw)          5000.0
set opt(bitrate)     4800.0
set opt(ack_mode)    "setNoAckMode"
```

In this part, we set up some variables on an associative array that will be useful later in the script. In particular, we set the number of nodes (4 in this case), the size of the packet (in bytes), the start time and the stop time for the traffic generation process (in seconds), the calculation of the duration of the traffic generation process (required, e.g., for the calculation of the average throughput), frequency, bandwidth and bitrate used at the PHY layer, and the acknowledgement setup at the mac layer.

```
set rng [new RNG]
set rng_position [new RNG]
```

With these two lines we set up a random number generator, useful for the positioning of the nodes.

```
set rnd_gen [new RandomVariable/Uniform]
$rnd_gen use-rng $rng
if {$opt(trace_files)} {
    set opt(tracefilename)      "./test_uwcsmaaloha.tr"
    set opt(tracefile) [open $opt(tracefilename) w]
    set opt(cltracefilename) "./test_uwcsmaaloha.cltr"
    set opt(cltracefile) [open $opt(tracefilename) w]
} else {
    set opt(tracefilename) "/dev/null"
    set opt(tracefile) [open $opt(tracefilename) w]
    set opt(cltracefilename) "/dev/null"
    set opt(cltracefile) [open $opt(cltracefilename) w]
}
```

With the code above we set up the name of the trace-file, which will contain a log of all the packet exchanges that take place among the modules in the nodes, during both the transmission and the reception phase. In the `.cltr` file, instead, all the cross-layer messages are traced. Here, again if the variable `opt(trace_files)` is set to 1, the file is defined, otherwise the output of the file is simply redirected to `/dev/null`, and no actual file gets written.

```
set channel [new Module/UnderwaterChannel]
set propagation [new MPropagation/Underwater]
set data_mask [new MSpectralMask/Rect]
$data_mask setFreq $opt(freq)
$data_mask setBandwidth $opt(bw)

Module/UW/CBR set packetSize_ $opt(pktsize)
Module/UW/CBR set period_     $opt(cbr_period)
Module/UW/CBR set PoissonTraffic_ 1
Module/UW/CBR set debug_          0

Module/MPhy/BPSK set TxPower_ $opt(txpower)
```

The lines above set up the underwater channel. In this case, we choose the `MPropagation/Underwater` module, that implements the Urick model, i.e., the empirical equations modeling the loss of acoustic power as caused by the superposition of the spreading and absorption losses. Together with the channel, the propagation model and the spectral mask of the PHY layer are also set. In particular, the spectral mask is a simple `Rect` function (which accepts as input the carrier frequency and the bandwdth), meaning that the system employs the whole bandwidth available. After that, we set up some variables for the CBR (short for *Constant Bit Rate*), which is an Application-layer module that generates data packets with a specific rate (specified by the user). The packet inter-arrival time can be either set to a deterministic value (specified by the user via `period_` variable) or modeled as a Poisson random variable (if the variable `PoissonTraffic_` is set to 1) where in this case the `period_` variable represents the average value

of the exponentially-distributed inter-arrival time. The variable `debug_` indicates whether or not the logs of the protocols should be written on the screen and it typically controls the amount of debug for all DESERT modules. After that, we set up the transmission power for the PHY layer, expressed in dB relative to 1 μPa^2 at 1 m from the source. If other protocols or modules need some initialization, here you can add all other parameters of each module. To see all the parameters of a specific module, please refer to the tcl file that you can find into the folder of that module.

2.4 Creating a node

```
proc createNode { id } {
    global channel propagation data_mask ns
    global cbr position node udp portnum ipr
    glogal ipif channel_estimator phy posdb opt
    global rvposx rvposy rvposz
    global mhrouting mll mac woss_utilities
    global node_coordinates woss_creator db_manager
```

Above we define all the global variables, or declare some variables already initialized, in order to let them be visible also inside the procedure (for more information on the scope of a variable, please refer to a tcl language guide).

```
set node($id) [$ns create-M_Node $opt(tracefile) $opt(cltracefile)]
for {set cnt 0} {$cnt < $opt(nn)} {incr cnt} {
    set cbr($id,$cnt)  [new Module/UW/CBR]
    set udp($id,$cnt)  [new Module/UW/UDP]
}
set ipr($id)  [new Module/UW/StaticRouting]
set ipif($id) [new Module/UW/IP]
set mll($id)  [new Module/UW/MLL]
set mac($id)  [new Module/UW/CSMA_ALOHA]
set phy($id)  [new Module/MPhy/BPSK]

for {set cnt 0} {$cnt < $opt(nn)} {incr cnt} {
    $node($id) addModule 7 $cbr($id,$cnt)   1  "CBR"
    $node($id) addModule 6 $udp($id,$cnt)   1  "UDP"
}
$node($id) addModule 5 $ipr($id)   1  "IPR"
$node($id) addModule 4 $ipif($id)  1  "IPF"
$node($id) addModule 3 $mll($id)   1  "MLL"
$node($id) addModule 2 $mac($id)   1  "MAC"
$node($id) addModule 1 $phy($id)   1  "PHY"

for {set cnt 0} {$cnt < $opt(nn)} {incr cnt} {
    $node($id) setConnection $cbr($id,$cnt)   $udp($id,$cnt)
    $node($id) setConnection $udp($id,$cnt)   $ipr($id)
    set portnum($id,$cnt) [$udp($id,$cnt) assignPort $cbr($id,$cnt) ]
```

```
}
$node($id) setConnection $ipr($id)   $ipif($id)  1
$node($id) setConnection $ipif($id)  $mll($id)   1
$node($id) setConnection $mll($id)   $mac($id)   1
$node($id) setConnection $mac($id)   $phy($id)   1
$node($id) addToChannel  $channel    $phy($id)   1

if {$id > 254} {
    puts "hostnum > 254!!! exiting"
    exit
}
```

The box above does several things. First of all, we declare all the protocols that we want to be part of the network protocol stack of the node. We build `opt(nn)` CBR and UDP layers, because we set up one Application and one UDP layer for each node in the network, in order to make a logical link from each Application Layer to one Application Layer of all the other nodes. After that, we use the method `addModule` to add all modules declared to the node. The method accepts as input the level of the module (useful to put all the modules in the correct order), the module itself, a flag that indicates if the event of reception or transmission of a packet from the upper or to the lower layers are traced down on a trace file, and also a brief string that identificate the layer in the trace file script. With the method `setConnection` instead, we connect together the various layers.

```
#Set the IP address of the node
set ip_value [expr $id + 1]
$ipif($id) addr $ip_value

set position($id) [new "Position/BM"]
$node($id) addPosition $position($id)
set posdb($id) [new "PlugIn/PositionDB"]
$node($id) addPlugin $posdb($id) 20 "PDB"
$posdb($id) addpos [$ipif($id) addr] $position($id)

#Setup positions
$position($id) setX_ [expr $id*200]
$position($id) setY_ [expr $id*200]
$position($id) setZ_ -100

#Interference model
set interf_data($id) [new "MInterference/MIV"]
$interf_data($id) set maxinterval_ $opt(maxinterval_)
$interf_data($id) set debug_       0

#Propagation model
$phy($id) setPropagation $propagation

$phy($id) setSpectralMask $data_mask
$phy($id) setInterference $interf_data($id)
```

```
$mac($id) $opt(ack_mode)
$mac($id) initialize
}
# -- this brace concludes the node configuration cycle

for {set id 0} {$id < $opt(nn)} {incr id}  {
    createNode $id
}
```

In the lines above we set up the IP address of the node, and the position of the node. Each position here is identified by a X, Y and Z coordinate, where the Z coordinate is negative and represents depth. After that, we define the interference model, which accepts as input the `maxinterval_` value, which is a sort of "memory" of the past interference power, and indicates for how long to keep trace of it. For what concerns the PHY layer, we now *assign* to the PHY layer all the necessary parameters. Namely, we assign the propagation model and the spectral mask (that we already specified before), as well as the interference model. We finally set up the acknowledgement mode of the MAC layer (in this case, we do not use ACK packets) and we initialize the protocol. After that, we call the procedure to create the desired number of nodes.

2.5 Setting up the connections among the nodes

```
proc connectNodes {id1 des1} {
  global ipif ipr portnum
  global cbr cbr_sink ipif_sink
  global portnum_sink ipr_sink opt

  $cbr($id1,$des1) set destAddr_ [$ipif($des1) addr]
  $cbr($id1,$des1) set destPort_ $portnum($des1,$id1)
  $cbr($des1,$id1) set destAddr_ [$ipif($id1) addr]
  $cbr($des1,$id1) set destPort_ $portnum($id1,$des1)
}

##################
# Setup flows    #
##################
for {set id1 0} {$id1 < $opt(nn)} {incr id1}  {
  for {set id2 0} {$id2 < $opt(nn)} {incr id2}  {
          connectNodes $id1 $id2
        }
}
```

In the code above, the CBR modules are linked together in order to set up the traffic flows among the node. In particular, we recall that each node has as many CBR modules as the number of nodes in the network. And whenever we link node i to node j, the jth CBR of node i will be configured to send packets to node j

(note that the CBR accepts as input the IP address of the destination and the port number associated to it) and vice-versa with the ith CBR module of node j. This association is performed within the procedure `connectNodes`, which is finally called for each pair of nodes in the network.

```
##################
# ARP tables     #
##################
for {set id1 0} {$id1 < $opt(nn)} {incr id1} {
    for {set id2 0} {$id2 < $opt(nn)} {incr id2} {
        $mll($id1) addentry [$ipif($id2) addr] [$mac($id2) addr]
    }
}
##################
# Routing tables #
##################
for {set id1 0} {$id1 < $opt(nn)} {incr id1}  {
    for {set id2 0} {$id2 < $opt(nn)} {incr id2}  {
        $ipr($id1) addRoute [$ipif($id2) addr] [$ipif($id2) addr]
    }
}
```

Above, we fill the ARP tables using the command `addentry` provided by the MLL layer, which accepts as input the IP address and the MAC layer of the node. Please note that the ARP tables have to be filled before starting the simulation, because the MLL layer will not provide ARP traffic in order to fill the ARP tables dinamically. We prefer this configuration because we do not want to add ARP control traffic to the network to discover MAC↔IP associations that are typically known in advance, especially in underwater networks. After that we fill the routing tables (necessary for a static routing protocol as in this case). The routing protocol used provides the command `addRoute` which accept as input the IP address of the destination and the IP address of next hop.

2.6 Traffic generation start and stop times

```
for {set id1 0} {$id1 < $opt(nn)} {incr id1}  {
        for {set id2 0} {$id2 < $opt(nn)} {incr id2} {
                if {$id1 != $id2} {
                        $ns at $opt(starttime) "$cbr($id1,$id2) start"
                        $ns at $opt(stoptime) "$cbr($id1,$id2) stop"
                }
        }
}
```

In this section we simply start and stop the traffic generation using the scheduler of ns and the start and stop commands provided by ns.

2.7 Finish procedure and start simulation

```
proc finish {} {
  global ns opt outfile
  global mac propagation
  global cbr_sink mac_sink phy_data
  global phy_data_sink channel
  global node_coordinates db_manager propagation
  global ipr_sink ipr ipif udp cbr phy phy_data_sink
  global node_stats tmp_node_stats sink_stats tmp_sink_stats
  if ($opt(verbose)) {
    puts "Simulation␣summary"
    puts "number␣of␣nodes␣␣:␣$opt(nn)"
    puts "packet␣size␣␣␣␣␣:␣$opt(pktsize)␣byte"
    puts "cbr␣period␣␣␣␣␣␣:␣$opt(cbr_period)␣s"
    puts "number␣of␣nodes␣␣:␣$opt(nn)"
    puts "simulation␣length:␣$opt(txduration)␣s"
    puts "tx␣power␣␣␣␣␣␣␣:␣$opt(txpower)␣dB"
    puts "tx␣frequency␣␣␣␣:␣$opt(freq)␣Hz"
    puts "tx␣bandwidth␣␣␣␣:␣$opt(bw)␣Hz"
    puts "bitrate␣␣␣␣␣␣␣␣:␣$opt(bitrate)␣bps"
  }
  set sum_cbr_throughput     0
  set sum_per                0
  set sum_cbr_sent_pkts      0.0
  set sum_cbr_rcv_pkts       0.0

  for {set i 0} {$i < $opt(nn)} {incr i}  {
        for {set j 0} {$j < $opt(nn)} {incr j} {
          set cbr_throughput [$cbr($i,$j) getthr]
                if {$i != $j} {
                  set cbr_sent_pkts [$cbr($i,$j) getsentpkts]
                        set cbr_rcv_pkts  [$cbr($i,$j) getrecvpkts]
                }
                if ($opt(verbose)) {
                  puts "cbr($i,$j)␣throughput:␣$cbr_throughput"
                }
        }
  set sum_cbr_throughput [expr $sum_cbr_throughput + $cbr_throughput]
  set sum_cbr_sent_pkts [expr $sum_cbr_sent_pkts + $cbr_sent_pkts]
  set sum_cbr_rcv_pkts [expr $sum_cbr_rcv_pkts + $cbr_rcv_pkts]
  }

  set ipheadersize [$ipif(1) getipheadersize]
  set udpheadersize [$udp(1,0) getudpheadersize]
  set cbrheadersize [$cbr(1,0) getcbrheadersize]

  if ($opt(verbose)) {
    puts "Mean␣Throughput␣:␣[expr␣($sum_cbr_throughput/(($opt(nn))*($opt(
        ↪ nn)-1)))]"
    puts "Sent␣Packets␣␣␣␣:␣$sum_cbr_sent_pkts"
    puts "Received␣Packets␣:␣$sum_cbr_rcv_pkts"
    puts "Packet␣Delivery␣Ratio:␣[expr␣$sum_cbr_rcv_pkts␣/␣
        ↪ $sum_cbr_sent_pkts␣*␣100]"
    puts "IP␣Pkt␣Header␣Size␣:␣$ipheadersize"
```

```
    puts "UDP Header Size : $udpheadersize"
    puts "CBR Header Size : $cbrheadersize"
    puts "done!"
  }

  $ns flush-trace
  close $opt(tracefile)
}
```

With the code above, we provide a procedure that is called as soon as the simulation is completed. This procedure uses the commands provided by the modules in order to calculate some metrics of interest, such as the throughput, the packet error rate, the total number of packets transmitted and received, together with a summary of the most important parameters of the simulation. At the end of the procedure, we close the tracefile.

```
$ns at [expr $opt(stoptime) + 250.0]  "finish; $ns halt"
$ns run
```

At the end of the script we schedule the end of simulation, by first calling the `finish` procedure and then by stopping ns via the command `halt`. After that, we start the simulation using the command `run` provided by ns.

3 Configuration for a real-life test

We now describe how to use DESERT to drive real modems, with focus on the S2C and WiSE EvoLogics modem series. To do so, we consider an experiment involving the Uw-Polling protocol. Of course, in order to run the following script and its related experiment, you need EvoLogics modems, powered and connected to your host via Ethernet cables. In the following paragraph we will describe the tcl and the `sh` script that we built in order to automate the insertion of the parameters into the tcl as much as possible. Rather than describing in details the tcl script (which is mostly equal to the script of Section 2), we will describe the novel script sections and their role.

3.1 The script `S2C_uwpolling_sample.tcl`

```
Module/UW/AL set PSDU 1400
Module/UW/AL set debug_ 0
Module/UW/AL set interframe_period 0.e1
Module/UW/AL set frame_set_validity 0

UW/AL/Packer set SRC_ID_Bits 8
UW/AL/Packer set PKT_ID_Bits 8
UW/AL/Packer set FRAME_OFFSET_Bits 15
UW/AL/Packer set M_BIT_Bits 1
UW/AL/Packer set DUMMY_CONTENT_Bits 0
UW/AL/Packer set debug_ 0

NS2/MAC/Uwpolling/Packer set t_in_Bits 16
NS2/MAC/Uwpolling/Packer set t_fin_Bits 16
NS2/MAC/Uwpolling/Packer set uid_TRIGGER_Bits 16
NS2/MAC/Uwpolling/Packer set id_polled_Bits 8
NS2/MAC/Uwpolling/Packer set backoff_time_Bits 16
NS2/MAC/Uwpolling/Packer set ts_Bits 16
NS2/MAC/Uwpolling/Packer set n_pkts_Bits 16
NS2/MAC/Uwpolling/Packer set uid_PROBE_Bits 16
NS2/MAC/Uwpolling/Packer set id_node_Bits 8
NS2/MAC/Uwpolling/Packer set uid_POLL_Bits 16
NS2/MAC/Uwpolling/Packer set debug_ 1
```

In the sectionm above, we set the Adaptation Layer (AL), which has been developed specifically to let the protocols in DESERT be tested in real scenarios with modems. In particular, the AL is stacked between the MAC layer and the interface to the modem (located at the PHY layer). The AL manages the conversion of packets from the ns2 data structures into a stream of bits suitable to be transmitted by the modem, and vice-versa. In addition, the AL automatically fragments the created bit stream whenever it does not fit within the Physical-layer Service Data Unit (PSDU) of the modem in use. (Note that the PSDU is also a parameter of the

AL module). The fragments are re-assembled automatically by the receiver's AL module whenever they are recognized as part of the same packet.

Every protocol that is part of the stack of the node has a `packer` associated to it. The packer is a submodule that helps the AL retrieve the bit stream related to the header of a specific protocol. In particular, above the AL we can locate the packer for the Uw-Polling protocol (`NS2/MAC/Uwpolling/Packer`). The parameters set how bits are dedicated to each field of the protocol header. For example, if one knows in advance that the number of nodes in a given experiment is less than 256, one can limit the size of the node ID to 8 bits, and correspondingly set the parameter `id_node_Bits` of the Uw-Polling packet to `8`. With similar considerations, we can compress the header as much as possible, with no loss of information. In any event, we remark that this procedure is valid only for integer field values, as the automatic bit size reduction procedure simply takes as many bits as indicated by the user, starting from the least significant bit of each field. Therefore, more complex field size reductions (e.g., the quantization of a `double` number down to a lower precision level) requires specific functions that must be implemented by the user inside the related packer.

Note that the AL also introduces a header, which is used to identify the fragments that are part of the same packet. Therefore, the AL also has a packer that serializes the AL header information into a bit stream. The parameters required by the AL module are:

- `PSDU`: Size of the PSDU in in bytes
- `debug_`: if set to 1, activates the printout of debug strings
- `interframe_period`: lag forced between the transmission of two subsequent fragments that are part of the same packet[1]
- `frame_set_validity`: validity of the frame set, i.e., amount of time for which a given frame set will be kept in the internal buffer of the AL while waiting for all fragments to be received.

After the packers are configured, in the `createNode` procedure we need to create the packers and to attach them to the main packer, which is will then be responsible to call the packer sequentially for each protocol header to be serialized (at the transmitter) or reconstructed from the bit stream (at the receiver). The code of this section is as follows. Note the difference between the `linkPacker` command,

[1]Some modems cannot accept subsequent transmission commands if they are not sufficiently apart. The `interframe_period` parameter makes it possible to satisfy this constraint, thereby avoiding that, e.g., issuing two very close transmission commands results in the second one being disregarded.

which links the AL module to its *own* packer, and the `addPacker` command, which adds additional packers to the queue of the packers called sequentially by the AL upon serialization or reconstruction. (The calling order is the same as the one used to add the packers.)

```
set packer_ [new UW/AL/Packer]

set packer_payload0 [new NS2/COMMON/Packer]
set packer_payload1 [new UW/IP/Packer]
set packer_payload2 [new NS2/MAC/Packer]
set packer_payload3 [new NS2/MAC/Uwpolling/Packer]
set packer_payload4 [new UW/UDP/Packer]
set packer_payload5 [new UW/CBR/Packer]

$packer_ addPacker $packer_payload0
$packer_ addPacker $packer_payload1
$packer_ addPacker $packer_payload2
$packer_ addPacker $packer_payload3
$packer_ addPacker $packer_payload4
$packer_ addPacker $packer_payload5

$uwal_ linkPacker $packer_

$uwal_ set nodeID $opt(node)
```

3.2 The script `S2C_EvoLogics_uwpolling_sample.sh`

In order to simplify and automate the process of running an experiment on multiple nodes, we built a shell script, `S2C_EvoLogics_uwpolling_sample.sh`, that properly launches the tcl script just described. This script must be called once for every modem that participates to the experiment. In particular the shell script accepts five parameters:

- **ID of the node**, also used as the node address: this number will be both the IP address and the MAC address of the node;
- **ID of the sink**: the address of the sink; this number must be equal to the ID of a node actually available in the network in order for Uw-Polling to work properly; when a number is specified, that node will act as the sink and all other nodes will know what is the address of the sink;
- **IP of the EvoLogics modem**: actual IP address of the modem, used by the modem in DESERT to connect to the modem send commands, and receive messages;
- **TCP port of the EvoLogics modem**: actual TCP port of the modem, also used by the modem interface in DESERT;

- **ID of the experiment**: used to differentiate the storage of the logs across different experiments, as the specified ID will be part of the name of log file.

These parameters are listed at screen by calling the script with the `--help` option.

```
nc -w4 -z ${3} ${4} > /dev/null
err_check=$?
if [ ${err_check} -eq 1 ]
then
  echo "The␣socket␣${1}:${2}␣is␣not␣active!␣Check␣the␣IP␣and␣the␣port␣
      ↪ associated!"
else
  rm -rf S2C_Evologics_Uwpolling.tr
  rm -rf MODEM_log_*
  rm -rf Uwpolling_AUV_*_${5}.out
  rm -rf Uwpolling_NODE_*_${5}.out
  ns S2C_uwpolling_sample.tcl $1 $2 5 360000 5 $5 $3 $4
fi
```

In the lines above, the script checks whether the IP and the port specified as input exist and are actually reachable. If the IP address is not reachable or the TCP port is not open, the script will return an error and will stop the experiment. Assuming that the socket with the EvoLogics modem is open and working, the script removes the tracefiles and the logs of the correspondent experiment ID. Finally, the script launches the tcl by providing giving all needed parameters. Some of these are the same input parameters of the sh script, but we can notice some additional ones:

- The time at which the traffic generation starts (set to 5 seconds in this script)
- The time at which the traffic generation stops (set to 3600 seconds in this script)
- The inter-arrival period of the traffic generation process (set to 5 seconds in this script).

This value is actually quite low in this configuration, as it is intended for a test with two nodes (one node and one sink). For networks composed of more than two nodes, it is suggested to balance the network load by setting higher value.

4 Underwater optical network simulator

In this section we will explain how to simulate an underwater optical network with ambient light noise. First, we will describe the `test_uwoptical_prop.tcl` sample, then we will explain the correct use of the ambient light noise Lookup Tables database.

4.1 Description of `test_uwoptical_prop.tcl`

Please note that we will not describe every part of the tcl script. We will analyze only the section where the optical communication is configured. All other parts are similar to what you can find in the other samples.

```
Module/UW/OPTICAL/PHY   set TxPower_                    $opt(txpower)
Module/UW/OPTICAL/PHY   set BitRate_                    $opt(bitrate)
Module/UW/OPTICAL/PHY   set AcquisitionThreshold_dB_    $opt(opt_acq_db)
Module/UW/OPTICAL/PHY   set Id_                         $opt(id)
Module/UW/OPTICAL/PHY   set Il_                         $opt(il)
Module/UW/OPTICAL/PHY   set R_                          $opt(shuntRes)
Module/UW/OPTICAL/PHY   set S_                          $opt(sensitivity)
Module/UW/OPTICAL/PHY   set T_                          $opt(temperatura)
Module/UW/OPTICAL/PHY   set Ar_                         $opt(rxArea)
Module/UW/OPTICAL/PHY   set debug_                      0

Module/UW/OPTICAL/Propagation set Ar_       $opt(rxArea)
Module/UW/OPTICAL/Propagation set At_       $opt(txArea)
Module/UW/OPTICAL/Propagation set c_        $opt(c)
Module/UW/OPTICAL/Propagation set theta_    $opt(theta)
Module/UW/OPTICAL/Propagation set debug_    0

set propagation [new Module/UW/OPTICAL/Propagation]
$propagation setOmnidirectional

set channel [new Module/UW/Optical/Channel]
```

Three modules are mainly involved in the simulation of an underwater optical network are: optical physical layer, propagation and channel.

`UW/OPTICAL/PHY` is the physical layer employed. It simulates the behavior of a SI PIN photo-diode receiver and a finite light source transmitter. The receiver settings can be retrieved from real devices data-sheets.

`UW/OPTICAL/Propagation` models the optical propagation underwater, taking into account of transmitter and receiver effective area (`At_` and `Ar_`), diverge angle (`theta_`) and water attenuation coefficient (`c_`). Transmitter and receiver informations can be found in real devices data-sheets. In this example, an omnidirectional transmitter is used.

`UW/Optical/Channel` is the optical underwater channel employed where the refractive index of the medium can be set (using the `RefractiveIndex_`

attribute), however, we used the default value.

```
set interf_data($id) [new "MInterference/MIV"]
$interf_data($id) set maxinterval_ $opt(maxinterval_)
$interf_data($id) set debug_       0

$phy($id) setInterference $interf_data($id)
$phy($id) setPropagation $propagation
$phy($id) setSpectralMask $data_mask
$phy($id) setLUTFileName "$opt(LUTpath)"
$phy($id) setLUTSeparator "␣"
$phy($id) useLUT
```

In the lines above we assign to the optical PHY layer all the necessary parameters. Namely, we assign the propagation model and the spectral mask, as well as the interference model. In addition, we assign also the optical ambient light noise Lookup Table parameters: file name (with relative path) and separator. We final enable the LUT use.

4.2 Description of optical database

Inside the folder `samples/desert_samples/dbs/optical_noise` we provide a database of optical ambient light noise lookup tables for different scenarios of location, solar zenith inclination, water column depth, scattering, absorption and attenuation coefficients. Each LUT contains two columns: water depth and solar irradiance. In each scenario folder there is an `info` file providing information about the partition of the attenuation coefficient (c) in absorption (a) and scattering (b) coefficient of the LUTs. Some additional informations concerning the water conditions are embedded in each LUT file name: e.g., the file `PcXX_depthYY.ascii` refers to a water column with maximum depth `YY` meters, with attenuation coefficient described by the `XX` pattern. For instance, in the `scenario3/Pc0.15a_depth40.ascii`, refers to a water column of 40 meters and the case `a` with attenuation coefficient of 0.15 m^{-1} (the combination of absorption and scattering is described in the info file).

All the ambient light noise LUTs have been retrieved by employing Hydrolight, a state of the art radiative transfer model.

5 ROV - controller system

In this section, we will explain how to simulate the behavior of a ROV moving towards the way points sent by a wireless remote control and monitoring its current position and speed. First, we will describe the `test_uwrov.tcl` sample, then we will explain how to import the way points from an external file. Both ROV and controller are in the application layer. `uwrov` is a DESERT Add-on, and it has to be installed in order to use it.

5.1 Description of `test_rov.tcl`

Please note that we will not describe every part of the tcl script. We will analyze only the section where the optical communication is configured. All other parts are similar to what you can find in the other samples.

```
Module/UW/ROV set packetSize_           $opt(ROV_pktsize)
Module/UW/ROV set period_               $opt(ROV_period)
Module/UW/ROV set debug_                0

Module/UW/ROV/CTR set packetSize_           $opt(CTR_pktsize)
Module/UW/ROV/CTR set debug_                0
```

In these lines we set both control and monitoring packet size. The controller sends consecutive way-points to the ROV, by using control packets. When the ROV receives a new way-point, it moves toward the new position. The ROV monitors its current position to the controller by sending monitoring packets. The ROV monitoring feature is a periodic process, thus, also the ROV packet period needs to be set like in the CBR application, described in Section 2.

```
set nodeCTR [$ns create-M_Node $opt(tracefile) $opt(cltracefile)]
set applicationCTR  [new Module/UW/ROV/CTR]
createNode $nodeCTR $applicationCTR 0
set nodeROV [$ns create-M_Node $opt(tracefile) $opt(cltracefile)]
set applicationROV  [new Module/UW/ROV]
createNode $nodeROV $applicationROV 1
```

With the the `createNode` function, we set and connect the network stack layers for both ROV and controller nodes. Their stack differs only for the application layer, passed as a function parameter, while the other layers are set like in tcl example presented in Section 2. The application layer employed to simulate the ROV behavior is `UW/ROV`, while the controller uses the `UW/ROV/CTR`.

```
    set position($id) [new "Position/SM"]
    $node addPosition $position($id)
    $position($id) setX_ [expr $id*5]
```

```
    $position($id) setY_ [expr $id*5]
    $position($id) setZ_ -10
    $application setPosition $position($id)
```

In addition, in the `createNode` function, we set the initial position of both the nodes. In particular, we employ the `SMPosition` position type, in order to move from the current position toward a way-point with constant speed. The position pointer is passed to the application layer, in order to allow the application to monitor and control the node position.

```
set outfile [open "test_uwrov_results.csv" "w"]
close $outfile
set fp [open $opt(waypoint_file) r]
set file_data [read $fp]
set data [split $file_data "\n"]
foreach line $data {
  if {[regexp {^(.*),(.*),(.*),(.*)$} $line -> t x y z]} {
    $ns at $t "update_and_check"
    $ns at $t "$applicationCTR␣sendPosition␣$x␣$y␣$z"
  }
}
$ns at $opt(starttime)    "$applicationROV␣start"
$ns at $opt(stoptime)     "$applicationROV␣stop"

proc update_and_check {} {
  global position applicationROV
  $position(1) update
  set outfile [open "test_uwrov_results.csv" "a"]
  puts $outfile "positions␣ROV:␣x␣=␣[$applicationROV␣getX],␣y␣=␣[
      ↪ $applicationROV␣getY],␣z␣=␣␣[$applicationROV␣getZ]"
  close $outfile
}
```

During a simulation run, the controller sends to the ROV command packets containing one way-point each. The packet transmission moment plus the way-point coordinates are obtained by reading the `waypoint_file`. As we will see in the next Section, we provide a database of way-point files for a well defined trajectory. Before to send a new way-point, the current actual position of the ROV is salved in the `outfile`, in order to permit post processing analysis.

5.2 Description of paths database

Inside the folder `samples/desert_samples/dbs/wp_path` we provide a database of way-point files for a well known path. Each path details is described in a `readme.txt` file. The path files are formatted as following:

- there are four columns, separated with a comma,
- the first column contains the simulated packet transmission time,

- the second column contains the x coordinate of the way-point position,
- the third column contains the y coordinate of the way-point position,
- the last column contains the z coordinate of the way-point position.

To create this files we payed attention on the following things: assuming an ROV that moves at constant speed (specified in the `readme.txt` file), each way-point is sent when the previous one is supposed to be achieved, plus a guard time `tg`. The value of `tg` is reported in each file name (e.g., `path_tg2s.csv` has a guard time of 2 s).