—DESERT Underwater v2.0.0 —

Description of a representative sample employing the WOSS libraries

1 Introduction

In this short guide, we will describe a representative sample that employs the DESERT Underwater framework in conjunction with the WOSS libraries. For more details on WOSS please refer to `http://telecom.dei.unipd.it/ns/woss/`. Please note that we will not describe every part of the tcl script. We will analyze only the section where WOSS is configured and interfaced with DESERT. All other parts are similar to what you can find in any tcl script within the `desert_samples` folder. For any problem, you can refer to the pdf guide that you can find in the same folder. Note that this is just a representative example, which employs some typical WOSS features, but does not exhaustively resort to all WOSS capabilities.

2 Description of `test_desert_woss_dbs.tcl`

This example makes it possible to WOSS for simulating realistic channel patterns and feed them to the DESERT Underwater libraries to simulate how the network reacts to these channel realizations. The computation of the interference among concurrent transmissions (as well as among replicas of the same transmission) takes full advantage of the capabilities of WOSS to exploit Bellhop to derive channel impulse responses by tracing different propagation paths. To do so, this tcl sample requires the use of environmental databases for SSP, bathymetry and bottom sediment. You can download the sediment and SSP databases from `http://telecom.dei.unipd.it/ns/woss/`, whereas for the GEBCO bathymetry database you have to register yourself and download it from `http://www.gebco.net/data_and_products/gridded_bathymetry_data/`.
You can find more info at
`http://telecom.dei.unipd.it/ns/woss/doxygen/installation.html`.
After the databases have been downloaded, please update the `opt(db_path)` and
`opt(db_res_path)` variables in the script with the actual database paths (see also below).

First of all, we set the tcl parameters that specify at which latitude and longitude in the world the network nodes will be located. In this case, we choose a location west of the Pianosa island, Italy, in the Tyrrhenian Sea. If you want to test a network, e.g., off the Portugal coast in front of Porto, you can set `start_lat` to 41.09 and `start_lon` to –8.7. In any event, be sure the coordinates you set are not on land!

```
set opt(start_lat) 42.59
set opt(start_long) 10.125
```

You have to set the path of the oceanographic databases just downloaded. You can set them via the following tcl parameters:

```
set opt(db_res_path) "."

set opt(db_path) "insert\_db_path_here"
set opt(db_path_gebco) "insert_gebco_path_here"
```

The variable `db_res_path` indicates where WOSS will store the channel realization it computes as the simulation time goes by. These realizations can be reused or not, depending on whether the user wants to simulate a time varying channel, whether or not the nodes are mobile, etc. For example, if the network is static and the channel realizations are not expected to change much over time, and one wants to perform several simulation runs in order to get smooth averages, it is useful to avoid the recalculation of the channel response for every simulation run. This folder defaults to the same path of the tcl script.

The WOSS framework requires several initializations. However, these can be left untouched to their default values for almost all purposes. Hereafter we list them with a brief description:

```
WOSS/Definitions/RandomGenerator/NS2 set rep_number_ $opt(rep_num)
WOSS/Definitions/RandomGenerator/C set seed_ $opt(rep_num)

set ssp_creator [new "WOSS/Definitions/SSP"]
set sediment_creator [new "WOSS/Definitions/Sediment"]
set pressure_creator [new "WOSS/Definitions/Pressure"]
set time_arr_creator [new "WOSS/Definitions/TimeArr"]
set time_reference [new "WOSS/Definitions/TimeReference/NS2"]
set transducer_creator [new "WOSS/Definitions/Transducer"]
set altimetry_creator [new "WOSS/Definitions/Altimetry/Bretschneider"]
set rand_generator [new "WOSS/Definitions/RandomGenerator/C"]
$rand_generator initialize
```

In the following box, we define and initialize the SSP, Sediment, Pressure, arrival time of the samples of the channel, transducer (if a model of a particular transducer is used), altimetry (surface waves model) and a random number generator used by WOSS framework.

```
set def_handler [new "WOSS/Definitions/Handler"]
$def_handler setSSPCreator $ssp_creator
$def_handler setSedimentCreator $sediment_creator
$def_handler setPressureCreator $pressure_creator
$def_handler setTimeArrCreator $time_arr_creator
$def_handler setTransducerCreator $transducer_creator
$def_handler setTimeReference $time_reference
$def_handler setRandomGenerator $rand_generator
$def_handler setAltimetryCreator $altimetry_creator
```

After the definition of the variables required by WOSS, in the box above we define the handlers, one for each variable defined before.

```
WOSS/Creator/Database/Textual/Results/TimeArr set debug 0
WOSS/Creator/Database/Textual/Results/TimeArr set woss_db_debug 0
WOSS/Creator/Database/Textual/Results/TimeArr set space_sampling 0.0

set db_res_arr [new "WOSS/Creator/Database/Textual/Results/TimeArr"]
$db_res_arr setDbPathName "${opt(db_res_path)}/test_aloha_with_dbs_res_arr.txt"
```

In the box above, the txt file holds all ray arrivals detected by WOSS for a given channel realization. The `TimeArr` object is responsible for storing and handling these arrivals. For example, its `space_sampling` parameter makes it possible to specify whether or not the same channel realization should be reused if the nodes move. By setting it to, e.g., 50 m, we specify that we want to compute a fresh channel realization for the link between a given transmitter and receiver whenever either moves more than 50 m away from the position they held when the last channel realization was computed. By setting it to 0 as in this example, we specify that we want a fresh channel realization every time regardless of how much the nodes move. Note that, in this script, we are simulating a static network with fixed environmental parameters, hence there is nothing that will make the channel vary over time.

```
WOSS/Creator/Database/NetCDF/Sediment/DECK41 set debug         0
WOSS/Creator/Database/NetCDF/Sediment/DECK41 set woss_db_debug 0

set db_sedim [new "WOSS/Creator/Database/NetCDF/Sediment/DECK41"]
$db_sedim setUpDeck41CoordinatesDb "${opt(db_path)}/seafloor_sediment/
    ↪ DECK41_coordinates.nc"
$db_sedim setUpDeck41MarsdenDb "${opt(db_path)}/seafloor_sediment/DECK41_mardsen_square.nc
    ↪ "
$db_sedim setUpDeck41MarsdenOneDb "${opt(db_path)}/seafloor_sediment/
    ↪ DECK41_mardsen_one_degree.nc"


WOSS/Creator/Database/NetCDF/SSP/WOA2005/MonthlyAverage set debug 0
WOSS/Creator/Database/NetCDF/SSP/WOA2005/MonthlyAverage set woss_db_debug 0

set db_ssp [new "WOSS/Creator/Database/NetCDF/SSP/WOA2005/MonthlyAverage"]
$db_ssp setDbPathName "${opt(db_path)}/ssp/2WOA2009_SSP_April.nc"


WOSS/Creator/Database/NetCDF/Bathymetry/GEBCO set debug           0
WOSS/Creator/Database/NetCDF/Bathymetry/GEBCO set woss_db_debug   0

set db_bathy [new "WOSS/Creator/Database/NetCDF/Bathymetry/GEBCO"]
$db_bathy setDbPathName "${opt(db_path_gebco)}/bathymetry/gebco_08.nc"
$db\_bathy useThirtySecondsPrecision
```

Above, we define the interfaces between the databases and WOSS. In particular, we define the interface with the GEBCO database, which is used to retrieve bathymetry data. We define also the interface with the databases where the sediment of the sea floor are specified and with the WOA database, which contains sound speed profile (SSP) data. We recall that the SSP depends on the season (hence on the month of the year). Here, for example, we chose to retrieve the average SSP profile computed over the month of April for the location set using the `start_lat` and `start_lon` parameters on page 2. To test the network with environmental data related a different month, you can change the month in `2WOA2009_SSP_April.nc`, e.g., try `2WOA2009_SSP_August.nc`.

```
WOSS/Creator/Bellhop set debug 0.0
WOSS/Creator/Bellhop set woss_debug 0.0
WOSS/Creator/Bellhop set evolution_time_quantum -1.0
WOSS/Creator/Bellhop set total_runs 5
WOSS/Creator/Bellhop set frequency_step 0.0
WOSS/Creator/Bellhop set total_range_steps 3000.0
WOSS/Creator/Bellhop set tx_min_depth_offset 0.0
WOSS/Creator/Bellhop set tx_max_depth_offset 0.0
WOSS/Creator/Bellhop set total_transmitters 1
WOSS/Creator/Bellhop set total_rx_depths 2
WOSS/Creator/Bellhop set rx_min_depth_offset -0.1
WOSS/Creator/Bellhop set rx_max_depth_offset 0.1
WOSS/Creator/Bellhop set total_rx_ranges 2
WOSS/Creator/Bellhop set rx_min_range_offset -0.1
WOSS/Creator/Bellhop set rx_max_range_offset 0.1
WOSS/Creator/Bellhop set total_rays 0.0
WOSS/Creator/Bellhop set min_angle -45.0
WOSS/Creator/Bellhop set max_angle 45.0
WOSS/Creator/Bellhop set ssp_depth_precision 1.0e-8
WOSS/Creator/Bellhop set normalized_ssp_depth_steps 100000


set woss_creator [new "WOSS/Creator/Bellhop"]
$woss_creator setWorkDirPath "/dev/shm/woss/aloha_with_dbs/"
$woss_creator setBellhopPath ""
$woss_creator setBellhopMode 0 0 "A"
$woss_creator setBeamOptions 0 0 "B"
$woss_creator setBathymetryType 0 0 "L"
$woss_creator setAltimetryType 0 0 "L"
$woss_creator setSimulationTimes 0 0 1 1 2013 0 0 1 2 1 2013 0 0 1
```

Above, we define the interface between WOSS and the Bellhop ray tracing software. Bellhop (see `http://oalib.hlsresearch.com/Rays/`) predicts acoustic pressure fields in an ocean water column by tracing the rays leaving a point source, singles out the rays that "hit" the receiver, and computes the acoustic pressure associated to each ray. The computation takes into account absorption, environmental parameters such as the SSP, the bathymetry and the bottom sediments, and optionally the transducer beam pattern and a snapshot of the surface wave shape. The output of Bellhop varies in accordance with the options set in its configuration file. Typically this is a set of arrivals, where each arrival corresponds to one ray traced from the source to the destination, and is characterized by an arrival delay and a complex amplitude.

In the box above, we also define all variables required to interface WOSS with Bellhop. We highlight a few ones in the following:

- the maximum and minimum angles of the source beam transmitted (this is equivalent to assuming an ideally flat acoustic beam emission pattern in the specified angular range, and no emission elsewhere)
- the total number of points (along the range dimension) where Bellhop will calculate the acoustic field
- how many points around the receiver position should be used to compute the pressure field; in this case, we are computing the field at two points, one 0.1 m ahead of the receiver location, and a second one 0.1 m beyond the receiver location; WOSS automatically derives a single arrival profile from the two profiles thus computed by Bellhop;
- whether we want to take int account the evolution of the acoustic field over time or not (e.g., because of mobility or of time-varying environmental properties)
- the number of rays to use (0 means automatic choice)

Furthermore, we define the path where WOSS can save the partial files used in the calculation (in this case, `/dev/shm/woss/aloha_with_dbs`), the path of the Bellhop binaries (in this case this is blank because the path of the binaries is part of the `$PATH` shell environment variable). Finally, with the command `setBellhopMode` we decide if the computation of the acoustic field should be done using the "incoherent" or the "coherent" mode (the former assumes that no knowledge of the ray phases, the latter assumes perfect knowledge). For more details, please take a look to the Bellhop guide at `http://oalib.hlsresearch.com/Rays/HLS-2010-1.pdf`

```
WOSS/PlugIn/ChannelEstimator set debug_ 0.0

WOSS/ChannelEstimator set debug_ 0.0
WOSS/ChannelEstimator set space_sampling_ 0.0
WOSS/ChannelEstimator set avg_coeff_ 0.5
set channel_estimator [ new "WOSS/ChannelEstimator"]


WOSS/Module/Channel set channel_eq_snr_threshold_db_0
WOSS/Module/Channel set channel_symbol_resolution_ 5e-3
WOSS/Module/Channel set channel_eq_time_ -1
WOSS/Module/Channel set debug_ 0

set channel [new "WOSS/Module/Channel"]
$channel setWossManager $woss_manager
$channel setChannelEstimator $channel_estimator

WOSS/MPropagation set debug_ 0
set propagation [new "WOSS/MPropagation"]
$propagation setWossManager $woss_manager


set data_mask [new "MSpectralMask/Rect"]
$data_mask setFreq $opt(freq)
$data_mask setBandwidth $opt(bw)
```

WOSS provides also a channel estimator, a channel module and a propagation model. In the tcl lines above you can find their definitions. In particular the channel has some parameters as seen in the box above, e.g., a threshold on the received SNR (used to pick the first arrival of a given transmission that stems sufficiently above noise). Unless you have specific requirements needs, these parameters can be left to their default values. The spectral mask is also defined (same as in the examples without WOSS).

In the following lines, instead, we show how to set the position of the nodes. In particular, the topology of the network can be set by defining the x, y and z coordinates. WOSS provides tools to transform the cartesian coordinates of the nodes into (latitude,longitude,depth) triples starting from the same reference point expressed in latitude and longitude (as set on page 2) and specifying a "bearing," (standardly defined as the angle between the "north" direction and the direction towards which one wants to move, expressed in degrees and counted in the clockwise direction. After having used the WOSS tools, one can employ the resulting latitude and longitude values to set the coordinates of the nodes via the `setLatitude`, `setLongitude`, `setAltitude_`commands.

```
set sink_depth 10

set curr_lat [$woss_utilities getLatfromDistBearing $opt(start_lat) $opt(start_long) 180.0
    ↪  100]
set curr_lon [$woss_utilities getLongfromDistBearing $opt(start_lat) $opt(start_long) 90.0
    ↪   100]

set position_sink [new "WOSS/Position"]
$node_sink addPosition $position_sink

$position_sink setLatitude_ $curr_lat
$position_sink setLongitude_ $curr_lon
$position_sink setAltitude_ [expr -1.0 * $sink_depth]
```

The remaining parts of the tcl are the same as in the simpler case of the simulations without WOSS: for these parts please refer to the guide you can find in the `desert_samples` folder.

3 The case with no oceanographic databases: the script `test_desert_woss_no_dbs.tcl`

The present folder contains another tcl file, `test_desert_woss_no_dbs.tcl`. This file is more or less equal to the one just described, except that the SSP and the bathymetry data are not retrieved from the databases, but defined respectively in the user-custom file called `ssp-test.txt` (that you can also find in the present folder) and inside the tcl script via the `setCustomBathymetry` command. Also, note that the sediment and the wave model can be defined in the tcl, using the corresponding command. In the tcl file you can find comments with more detailed information.

4 A full case with DESERT, WOSS, mobile networks and time-varying environmental parameters: description of `test_woss_waypoints_time_evo.tcl`

The following example is a more complex version of the previous example. This script configures a mobile node, e.g., an AUV, to patrol a cluster of 4 nodes located on the sea bottom. As before, we proceed by highlighting the main differences with the previous scripts.

In the code snippet below, we configure the altimetry object that will be used by the framework. We set up the Bretschneider model with a characteristic wave height of 1 m and an average period of 1 s. The model will evolve every 100 seconds.

```
#each object will evolve only if 100 seconds has passed
WOSS/Definitions/Altimetry/Bretschneider set evolution_time_quantum   100
#### no need to set these two values, they will be binded by the WOSS object
WOSS/Definitions/Altimetry/Bretschneider set range                    -1
WOSS/Definitions/Altimetry/Bretschneider set total_range_steps        -1
WOSS/Definitions/Altimetry/Bretschneider set characteristic_height    1.0
WOSS/Definitions/Altimetry/Bretschneider set average_period           1.0
set cust_altimetry   [new "WOSS/Definitions/Altimetry/Bretschneider"]
```

We then create a time-evolving SSP model, again used by the whole framework. We provide three SSPs each in a different ASCII file, and set them to occur over a 5 h time span, which is more than enough to cover the ∼3 h of simulated time. For each simulated time epoch, WOSS will produce a different SSP as a convex combination of the two SSPs instances set by the user that occur closest in time. For example, a transmission occurring at 9:01 am will observe an SSP that is the point-wise average of the SSP set at 8:01 am and of the SSP set at 10:01 am.

```
#### We set different SSP for different Time values, to account for time
#### evolution
set time_evo_ssp_1 [new "WOSS/Definitions/Time"]
set time_evo_ssp_2 [new "WOSS/Definitions/Time"]
set time_evo_ssp_3 [new "WOSS/Definitions/Time"]

#first ssp, time key is 1st january 2014, 8:01 am
$time_evo_ssp_1 setTime 1 1 2014 8 0 1
#first ssp, time key is 1st january 2014, 10:01 am
$time_evo_ssp_2 setTime 1 1 2014 10 0 1
#first ssp, time key is 1st january 2014, 13:01 am
$time_evo_ssp_3 setTime 1 1 2014 13 0 1
```

We now plug the SSPs in the framework as follows:

```
set db_manager [new "WOSS/Database/Manager"]
$db_manager setCustomSediment   "Test Sedim" 1560 200 1.5 0.9 0.8 1.0
$db_manager setCustomAltimetry  $cust_altimetry
#we insert in the custom SSP database, each SSP value with its related Time key
$db_manager setCustomSSP        $time_evo_ssp_1 "./ssp-test.txt"
$db_manager setCustomSSP        $time_evo_ssp_2 "./ssp-test_2.txt"
$db_manager setCustomSSP        $time_evo_ssp_3 "./ssp-test_3.txt"
```

We import the characteristics of a real electro-acoustic transducer via the following code. In particular, we choose the ITC-3001 conical beam pattern transducer which has a central frequency of 17.5 kHz, the same used by the BPSK physical layer.

```
#### we import a transducer and we link it to "ITC-3001" tag
$transducer_handler importAscii "ITC-3001" "$opt(db_path)/transducers/ITC/
    ↪ ITC-ITC-3001-17.5kHz.txt"
```

We finally set up the AUV movement pattern using the waypoint mobility model. We set the time evolution quantum in seconds (i.e., the location of the AUV will be updated once every second of simulated time). Note that we also set the initial vertical orientation from the horizontal axis in decimal degrees for the AUV. The latter will vary between the the maximum and minimum vertical orientation if the depth of the waypoints changes.

```
#we set the time evolution quantum and the comparison distance
WOSS/Position/WayPoint set time_threshold_            [expr 1.0 / $opt(speed)]
WOSS/Position/WayPoint set compDistance_              0.0
WOSS/Position/WayPoint set verticalOrientation_       0.0
WOSS/Position/WayPoint set minVerticalOrientation_    -40.0
WOSS/Position/WayPoint set maxVerticalOrientation_    40.0
```

For each network node, we set up the transducer, within the `createNode` function. The initial vertical rotation from the horizontal axis is expressed in decimal degrees. A negative value means that the transducer is oriented towards the surface.

```
  #### we set the transducer labeled "ITC-3001" for the tx woss::Location and for all rx
      ↪ woss::Locations.
  #### Initial rotation is -45.0 , multiply costant is 1, add costant is 0
  $woss_creator setCustomTransducerType $position($id) 0 "ITC-3001" -45.0 0.0 1.0 0.0
...
  #### we set the transducer labeled "ITC-3001" for SPL, input power and energy consumpion
      ↪  computations
  $phy_data($id) setTransducerType  [expr [$data_mask getFreq] - [$data_mask getBandwidth]
      ↪  / 2.0 ] [expr [$data_mask getFreq] + [$data_mask getBandwidth] / 2.0 ] "ITC-3001
      ↪ "
```

We do the same with the AUV in the `createAUV` function. Note that we do not set the intial vertical rotation of the transducer for the AUV, i.e., the transducer of the AUV is set to be oriented horizontally, pointing towards the bearing direction.

We now create a custom bathymetry grid with the `createBathymetryMap` function. We define a grid from left to right and from the top left coordinates point by keeping the same longitude, and the latitude down (180° bearing). For each left-right grid line the depth starts from the 40 m and gradually increases to 240 m at each step.

```
proc createBathymetryMap { } {
  global db_manager woss_utilities opt rbathy

  set long [ $woss_utilities getLongfromDistBearing $opt(start_lat) $opt(start_long) -90.0
      ↪  500.0 ]
  set bathy_start 40.0
  set bathy_end   240.0
  set line_range  250.0

  set bathy_incr [ expr ($bathy_end - $bathy_start) / $line_range ]

  for { set bid 0.0 } { $bid < 2500.0 } { set bid [expr $bid + 10.0]} {
    set lat [ $woss_utilities getLatfromDistBearing $opt(start_lat) $long 180.0 $bid ]

#     puts "createBathymetryMap latitude = $lat ; longitude = $long"

    for { set bid2 0.0 } { $bid2 < 250.0 } { set bid2 [expr $bid2 + 10.0] } {
      set rb [$rbathy value]
      $db_manager setCustomBathymetry $lat $opt(start_long) 90.0 1 [expr 10.0 * $bid2] [
          ↪ expr $bathy_start + $bid2 * $bathy_incr + $rb ]
    }
  }
}
```

We finally set up the AUV waypoints. For each way point we define the latitude, the longitude, the depth and the speed. The cumulative time of arrival in seconds is returned by the `addWayPoint` function.

```
#### we set the AUV waypoints and set final loop point (3 loops)
proc createAUVWaypoints { } {
  global position_auv opt position woss_utilities rdepth
  set toa 0.0
  set curr_lat   [ $position(0) getLatitude_]
  set curr_lon   [ $position(0) getLongitude_]
  set curr_depth [expr -1.0 * $opt(auv_depth) * [$rdepth value]]
  set toa        [$position_auv addWayPoint $curr_lat $curr_lon $curr_depth $opt(speed) 0
       ↪ .0 ]
  puts "waypoint 1 lat = $curr_lat ; lon = $curr_lon ; depth = $curr_depth; toa = $toa"

  set curr_lat   [ $position(1) getLatitude_]
  set curr_lon   [ $position(1) getLongitude_]
  set curr_depth [expr -1.0 * $opt(auv_depth) * [$rdepth value]]
  set toa        [$position_auv addWayPoint $curr_lat $curr_lon $curr_depth $opt(speed) 0
       ↪ .0 ]
  puts "waypoint 2 lat = $curr_lat ; lon = $curr_lon ; depth = $curr_depth; toa = $toa"

  set curr_lat   [ $position(3) getLatitude_]
  set curr_lon   [ $position(3) getLongitude_]
  set curr_depth [expr -1.0 * $opt(auv_depth) * [$rdepth value]]
  set toa        [$position_auv addWayPoint $curr_lat $curr_lon $curr_depth $opt(speed) 0
       ↪ .0 ]
  puts "waypoint 3 lat = $curr_lat ; lon = $curr_lon ; depth = $curr_depth; toa = $toa"

  set curr_lat   [ $position(2) getLatitude_]
  set curr_lon   [ $position(2) getLongitude_]
  set curr_depth [expr -1.0 * $opt(auv_depth) * [$rdepth value]]
  set toa        [$position_auv addLoopPoint $curr_lat $curr_lon [expr -1.0 * $opt(
       ↪ auv_depth)] $opt(speed) 0.0 0 4 ]
  puts "waypoint 4 lat = $curr_lat ; lon = $curr_lon ; depth = $curr_depth; toa = $toa"
}
```

5 Conclusions

This concludes the presentation of three examples that showcase the integration between DESERT Underwater and WOSS. We encourage the reader to play with these examples, change some parameters, change the network topology and configuration, experiment with different protocols, and of course to write new protocols and test them using our framework.
We take this chance to thank the reader for his/her interest in DESERT Underwater and WOSS.