|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | | | |
|  | | Home Assignment: PHY Layer | | | | |  | |
|  |  | | | | | | |  |
|  | | | |  | |  | | |
|  | | | | Md Abu Ahammed Babu GENIAL Cohort 1 | |  | | |
|  | | | | 06/09/2021—Advanced Wireless Networks—Teacher: Evgeny Osipov | |  | | |
|  | | |  | |  | | | |

|  |
| --- |
| Q1. What are alternatives for installing NS3 under different operating systems? **Ans:** There are three (3) different alternatives to install NS-3 under different operating systems. These are:  **i) Installation using Bake:** Bake is a tool to installing, building and finding out the missing requirements for installing ns-3. It requires python 2.7 or above and git installed in the machine. Before installing ns-3 using bake, it needs to be configured by telling which module of ns-3 will be added. For example: to configure ns-3 using the version ns-3.34, the command to execute is:  *bake.py configure –e –ns-3.34*  To see the modules and specific system requirements for this configuration:  *bake.py show*  To download the modules, build and install:  *bake.py deploy*  **ii) Manual installation using ns-3-allinone:** ns-3-allinone is an environment which contains a set of scripts to manage downloading and building of various ns-3 subsystems from mercurial repositories using git. It is recommended to make a new repository in home directory to keep local git repositories.  The download.py file inside ns-3-allinone will help to download specific version of ns-3 distribution.  **iii) Manual installation using Tarball:** The process for downloading ns-3 via tarball is simpler than the Mercurial process since all of the pieces are pre-packaged. One just have to pick a release, download it and decompress it. |
| **Q2.** Write a step-by-step instruction for creating a simulation scenario, i.e. first we create ....  **Ans:** The step-by-step instructions for creating a simulation scenario could be like this:  1. Creating the nodes required for the simulation.  2. Defining the applications to be run on the nodes.  3. Creating particular communication channel (csma, wifi or point-to-point).  4. Assigning the IP addresses, MAC addresses of each connecting points of each devices inside nodes.  5. Managing the network topology using topology helper objects to connect several networks together. |
| **Q3.** If you would need to simulate a protocol which is not inside the NS3 library, what would you need to do?  **Ans:** There are several steps for adding a new protocol module inside NS-3 library. These are:  **Step 1: Creating the module skeleton**  To create a module named “new-module”, the command below needs to be run from the top directory  *./utils/create-module.py new-module*  By default, create-module.py creates the module skeleton in the src directory. However, it can also create modules in contrib:  *./utils/create-module.py contrib/new-contrib*  However, the create-module.py script will create the directories as well as initial skeleton wscript, .h, .cc and .rst files.  Then the module dependencies should be specified in the “wscript” file of the “module” directory. For example: to add internet, mobility and aodv module dependencies on the new module, the wscript will look like this:  **def** build(bld):  module = bld.create\_ns3\_module('new-module', ['internet', 'mobility', 'aodv'])  Finally, it is good practice to write tests and examples. These will almost certainly be required for new modules to be accepted into the official ns-3 source tree. A skeleton test suite and test case is created in the test/ directory. The skeleton test suite will contain the below constructor, which declares a new unit test named new-module, with a single test case consisting of the class NewModuleTestCase1:  NewModuleTestSuite::NewModuleTestSuite ()  : TestSuite ("new-module", UNIT)  {  AddTestCase (**new** NewModuleTestCase1);  }  **Step 2: Declare Source Files**  The public header and source code files for the new module should be specified in the wscript file by modifying it with a text editor.  As an example, after declaring the spectrum module, the src/spectrum/wscript specifies the source code files with the following list:  **def** build(bld):  module = bld.create\_ns3\_module('spectrum', ['internet', 'propagation', 'antenna', 'applications'])  module.source = [  'model/spectrum-model.cc',  'model/spectrum-value.cc',  .  .  .  'model/microwave-oven-spectrum-value-helper.cc',  'helper/spectrum-helper.cc',  'helper/adhoc-aloha-noack-ideal-phy-helper.cc',  'helper/waveform-generator-helper.cc',  'helper/spectrum-analyzer-helper.cc',  ]  The objects resulting from compiling these sources will be assembled into a link library, which will be linked to any programs relying on this module.  **Step 3: Declare Public Header Files**  The header files defining the public API of the new model and helpers also should be specified in the wscript file.  Continuing with the spectrum model illustration, the public header files are specified with the following stanza. (Note that the argument to the bld function tells waf to install this module’s headers with the other ns-3 headers):  headers = bld(features='ns3header')  headers.module = 'spectrum'  headers.source = [  'model/spectrum-model.h',  'model/spectrum-value.h',  .  .  .  'model/microwave-oven-spectrum-value-helper.h',  'helper/spectrum-helper.h',  'helper/adhoc-aloha-noack-ideal-phy-helper.h',  'helper/waveform-generator-helper.h',  'helper/spectrum-analyzer-helper.h',  ]  Headers made public in this way will be accessible to users of the new model with include statements like:  #include *"ns3/spectrum-model.h"*  **Step 4: Declare Tests**  If new module has tests, then they must be specified in the wscript file by modifying it with text editor.  The spectrum model tests are specified with the following stanza:  module\_test = bld.create\_ns3\_module\_test\_library('spectrum')  module\_test.source = [  'test/spectrum-interference-test.cc',  'test/spectrum-value-test.cc',  ]  **Step 5: Declare Examples**  If new module has examples, then they must be specified in the examples/wscript file. (The skeleton top-level wscript will recursively include examples/wscript only if the examples were enabled at configure time.)  The spectrum model defines it’s first example in src/spectrum/examples/wscript with:  **def** build(bld):  obj = bld.create\_ns3\_program('adhoc-aloha-ideal-phy',  ['spectrum', 'mobility'])  obj.source = 'adhoc-aloha-ideal-phy.cc'  **Step 6: Examples Run as Tests**  In addition to running explicit test code, the test framework can also be instrumented to run full example programs to try to catch regressions in the examples. However, not all examples are suitable for regression tests. The file test/examples-to-run.py controls the invocation of the examples when the test framework runs.  The spectrum model examples run by test.py are specified in src/spectrum/test/examples-to-run.py using the following two lists of C++ and Python examples:  *# A list of C++ examples to run in order to ensure that they remain*  *# buildable and runnable over time. Each tuple in the list contains*  *#*  *# (example\_name, do\_run, do\_valgrind\_run).*  *#*  *# See test.py for more information.*  cpp\_examples = [  ("adhoc-aloha-ideal-phy", "True", "True"),  ("adhoc-aloha-ideal-phy-with-microwave-oven", "True", "True"),  ("adhoc-aloha-ideal-phy-matrix-propagation-loss-model", "True", "True"),  ]  *# A list of Python examples to run in order to ensure that they remain*  *# runnable over time. Each tuple in the list contains*  *#*  *# (example\_name, do\_run).*  *#*  *# See test.py for more information.*  python\_examples = [  ("sample-simulator.py", "True"),  ]  **Step 7: Configure and Build**  Now the new module can be configured, built and tested as normal. The project must be reconfigured as a first step so that waf caches the new information in wscript files, or else new module will not be included in the build.  $ ./waf configure --enable-examples --enable-tests  $ ./waf build  $ ./test.py  **Step 8: Python Bindings**  Adding Python bindings to new module is optional, and the step is commented out by default in the create-module.py script.  *# bld.ns3\_python\_bindings()*  To include Python bindings (needed only if you want to write Python ns-3 programs instead of C++ ns-3 programs), The above line should be uncommented and install the Python API scanning system and scan the module to generate new bindings. |
| **Q4.** For compiling an ns-3 executable a special build system is used. What is this system?  **Ans:** The build system Waf is used to compile an ns-3 executable. It is one of the new generation of Python-based build systems. |
| **Q5.** Describe the purpose of different (only main) folders of the ns3 distribution.  **Ans:** The main folders of the ns-3 distribution are: src, scratch and examples. The src folder contains all the required header files, scratch folder contains the codes to run and the examples folder contains the default example codes for checking purpose. Besides these, contrib folder contains wscript, utils folder contains the various utility scripts and waf-tools folder contains the scripts required by waf for compilation. |
| **Q6.** Which folder should contain your simulation scripts?  **Ans:** The scratch folder should contain all the simulation scripts. For simulations consist of more than one scripts should be together in a sub-directory inside scratch folder. |
| **Q7.** Write a step-by step instruction for executing an ns-3 simulation.  **Ans:** To execute a ns-3 simulation using waf, simply use the –run command. For example: to run the hello world program, type:  *./waf –run hello-simulator*  To feed command line arguments during running, add another command –command-template  *./waf –run hello-simulator –command-template=”%s <args>”*  To run ns-3 programs under the control of another utility, such as a debugger (e.g. gdb) or memory checker (e.g. valgrind), use a similar --command-template="..." form.  For example, to run the default ns-3 program hello-simulator with the arguments <args> under the gdb debugger:  *./waf --run=hello-simulator --command-template="gdb %s --args "* |
| **Q8.** In how many formats does ns-3 saves the results(traces) of a simulation? Name them. What are the major differences?  **Ans:** ns-3 can save the traces of simulation in two different formats. One is called ASCII tracing which saves the tracing results in .tr format and the other is PCAP tracing which saves the results in .pcap format. The acronym pcap stands for packet capture.  The major difference between this two formats is: .pcap is actually an API that can be read by many traffic trace analyzer such as Wireshark which can provide better understanding with the help of graphical user interface which is not possible for ASCII tracing .tr formats. |
| **Q9.** When you run your simulation in which folder you will find the simulation traces?  **Ans:** After running a simulation, the traces will be saved at the top-level repository of the directory by default. However, it could be changed using –cwd option of waf. |