

ns-3 Testing and Validation

ns-3 project

feedback: ns-developers@isi.edu

7 September 2009

This is an ns-3 reference manual. Primary documentation for the ns-3 project is available in four forms:

- [ns-3 Doxygen](#): Documentation of the public APIs of the simulator
- [ns-3 Tutorial](#)
- [ns-3 Manual](#)
- Testing and Validation (this document)
- [ns-3 wiki](#)

This document is written in GNU Texinfo and is to be maintained in revision control on the ns-3 code server. Both PDF and HTML versions should be available on the server. Changes to the document should be discussed on the ns-developers@isi.edu mailing list.

This software is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This software is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

Table of Contents

1	Overview	1
2	Background	2
2.1	Correctness	2
2.2	Validation and Verification	2
2.3	Robustness	3
2.4	Performant	4
2.5	Maintainability	4
3	Testing Framework	6
3.1	Buildbots	6
3.2	Test.py	6
3.3	Test Taxonomy	10
3.3.1	Build Verification Tests	10
3.3.2	Unit Tests	10
3.3.3	System Tests	10
3.3.4	Examples	10
3.3.5	Performance Tests	11
3.4	Running Tests	11
3.5	Running Tests Under the Test Runner Executable	11
3.6	Class TestRunner	12
3.7	Test Suite	13
3.8	Test Case	13
3.9	Utilities	14
4	How to write tests	15
5	Propagation Loss Models	16
5.1	FriisPropagationLossModel	16
5.1.1	Model reference	16
5.1.2	Validation test	16
5.1.3	Discussion	17
5.2	LogDistancePropagationLossModel	17
5.2.1	Model reference	17
5.2.2	Validation test	17
5.2.3	Discussion	18

1 Overview

This document is concerned with the testing and validation of **ns-3** software.

This document provides

- a description of the ns-3 testing framework;
- a guide to model developers or new model contributors for how to write tests;
- validation and verification results reported to date.

2 Background

Writing defect-free software is a difficult proposition. There are many dimensions to the problem and there is much confusion regarding what is meant by different terms in different contexts. We have found it worthwhile to spend a little time reviewing the subject and defining some terms.

Software testing may be loosely defined as the process of executing a program with the intent of finding errors. When one enters a discussion regarding software testing, it quickly becomes apparent that there are many distinct mind-sets with which one can approach the subject.

For example, one could break the process into broad functional categories like “correctness testing,” “performance testing,” “robustness testing” and “security testing.” Another way to look at the problem is by life-cycle: “requirements testing,” “design testing,” “acceptance testing,” and “maintenance testing.” Yet another view is by the scope of the tested system. In this case one may speak of “unit testing,” “component testing,” “integration testing,” and “system testing.” These terms are also not standardized in any way, and so “maintenance testing” and “regression testing” may be heard interchangeably. Additionally, these terms are often misused.

There are also a number of different philosophical approaches to software testing. For example, some organizations advocate writing test programs before actually implementing the desired software, yielding “test-driven development.” Some organizations advocate testing from a customer perspective as soon as possible, following a parallel with the agile development process: “test early and test often.” This is sometimes called “agile testing.” It seems that there is at least one approach to testing for every development methodology.

The **ns-3** project is not in the business of advocating for any one of these processes, but the project as a whole has requirements that help inform the test process.

Like all major software products, **ns-3** has a number of qualities that must be present for the product to succeed. From a testing perspective, some of these qualities that must be addressed are that **ns-3** must be “correct,” “robust,” “performant” and “maintainable.” Ideally there should be metrics for each of these dimensions that are checked by the tests to identify when the product fails to meet its expectations / requirements.

2.1 Correctness

The essential purpose of testing is to determine that a piece of software behaves “correctly.” For **ns-3** this means that if we simulate something, the simulation should faithfully represent some physical entity or process to a specified accuracy and precision.

It turns out that there are two perspectives from which one can view correctness. Verifying that a particular process is implemented according to its specification is generically called verification. The process of deciding that the specification is correct is generically called validation.

2.2 Validation and Verification

A computer model is a mathematical or logical representation of something. It can represent a vehicle, a frog or a networking card. Models can also represent processes such as global

warming, freeway traffic flow or a specification of a networking protocol. Models can be completely faithful representations of a logical process specification, but they necessarily can never completely simulate a physical object or process. In most cases, a number of simplifications are made to the model to make simulation computationally tractable.

Every model has a *target system* that it is attempting to simulate. The first step in creating a simulation model is to identify this target system and the level of detail and accuracy that the simulation is desired to reproduce. In the case of a logical process, the target system may be identified as “TCP as defined by RFC 793.” In this case, it will probably be desirable to create a model that completely and faithfully reproduces RFC 793. In the case of a physical process this will not be possible. If, for example, you would like to simulate a wireless networking card, you may determine that you need, “an accurate MAC-level implementation of the 802.11 specification and [...] a not-so-slow PHY-level model of the 802.11a specification.”

Once this is done, one can develop an abstract model of the target system. This is typically an exercise in managing the tradeoffs between complexity, resource requirements and accuracy. The process of developing an abstract model has been called *model qualification* in the literature. In the case of a TCP protocol, this process results in a design for a collection of objects, interactions and behaviors that will fully implement RFC 793 in ns-3. In the case of the wireless card, this process results in a number of tradeoffs to allow the physical layer to be simulated and the design of a network device and channel for ns-3, along with the desired objects, interactions and behaviors.

This abstract model is then developed into an ns-3 model that implements the abstract model as a computer program. The process of getting the implementation to agree with the abstract model is called *model verification* in the literature.

The process so far is open loop. What remains is to make a determination that a given ns-3 model has some connection to some reality – that a model is an accurate representation of a real system, whether a logical process or a physical entity.

If one is going to use a simulation model to try and predict how some real system is going to behave, there must be some reason to believe your results – i.e., can one trust that an inference made from the model translates into a correct prediction for the real system. The process of getting the ns-3 model behavior to agree with the desired target system behavior as defined by the model qualification process is called *model validation* in the literature. In the case of a TCP implementation, you may want to compare the behavior of your ns-3 TCP model to some reference implementation in order to validate your model. In the case of a wireless physical layer simulation, you may want to compare the behavior of your model to that of real hardware in a controlled setting,

The ns-3 testing environment provides tools to allow for both model validation and testing, and encourages the publication of validation results.

2.3 Robustness

Robustness is the quality of being able to withstand stresses, or changes in environments, inputs or calculations, etc. A system or design is “robust” if it can deal with such changes with minimal loss of functionality.

This kind of testing is usually done with a particular focus. For example, the system as a whole can be run on many different system configurations to demonstrate that it can perform correctly in a large number of environments.

The system can also be stressed by operating close to or beyond capacity by generating or simulating resource exhaustion of various kinds. This genre of testing is called “stress testing.”

The system and its components may be exposed to so-called “clean tests” that demonstrate a positive result – that is that the system operates correctly in response to a large variation of expected configurations.

The system and its components may also be exposed to “dirty tests” which provide inputs outside the expected range. For example, if a module expects a zero-terminated string representation of an integer, a dirty test might provide an unterminated string of random characters to verify that the system does not crash as a result of this unexpected input. Unfortunately, detecting such “dirty” input and taking preventive measures to ensure the system does not fail catastrophically can require a huge amount of development overhead. In order to reduce development time, a decision was taken early on in the project to minimize the amount of parameter validation and error handling in the `ns-3` codebase. For this reason, we do not spend much time on dirty testing – it would just uncover the results of the design decision we know we took.

We do want to demonstrate that `ns-3` software does work across some set of conditions. We borrow a couple of definitions to narrow this down a bit. The *domain of applicability* is a set of prescribed conditions for which the model has been tested, compared against reality to the extent possible, and judged suitable for use. The *range of accuracy* is an agreement between the computerized model and reality within a domain of applicability.

The `ns-3` testing environment provides tools to allow for setting up and running test environments over multiple systems (buildbot) and provides classes to encourage clean tests to verify the operation of the system over the expected “domain of applicability” and “range of accuracy.”

2.4 Performant

Okay, “performant” isn’t a real English word. It is, however, a very concise neologism that is quite often used to describe what we want `ns-3` to be: powerful and fast enough to get the job done.

This is really about the broad subject of software performance testing. One of the key things that is done is to compare two systems to find which performs better (cf benchmarks). This is used to demonstrate that, for example, `ns-3` can perform a basic kind of simulation at least as fast as a competing product, or can be used to identify parts of the system that perform badly.

In the `ns-3` test framework, we provide support for timing various kinds of tests.

2.5 Maintainability

A software product must be maintainable. This is, again, a very broad statement, but a testing framework can help with the task. Once a model has been developed, validated and

verified, we can repeatedly execute the suite of tests for the entire system to ensure that it remains valid and verified over its lifetime.

When a feature stops functioning as intended after some kind of change to the system is integrated, it is called generically a regression. Originally the term regression referred to a change that caused a previously fixed bug to reappear, but the term has evolved to describe any kind of change that breaks existing functionality. There are many kinds of regressions that may occur in practice.

A *local regression* is one in which a change affects the changed component directly. For example, if a component is modified to allocate and free memory but stale pointers are used, the component itself fails.

A *remote regression* is one in which a change to one component breaks functionality in another component. This reflects violation of an implied but possibly unrecognized contract between components.

An *unmasked regression* is one that creates a situation where a previously existing bug that had no affect is suddenly exposed in the system. This may be as simple as exercising a code path for the first time.

A *performance regression* is one that causes the performance requirements of the system to be violated. For example, doing some work in a low level function that may be repeated large numbers of times may suddenly render the system unusable from certain perspectives.

The `ns-3` testing framework provides tools for automating the process used to validate and verify the code in nightly test suites to help quickly identify possible regressions.

3 Testing Framework

3.1 Buildbots

The `ns-3` testing framework is composed of several major pieces. At the highest level are the buildbots (build robots). If you are unfamiliar with this system look at <http://djmitche.github.com/buildbot/docs/0.7.11/>. This is an open-source automated system that allows `ns-3` to be rebuilt and tested each time something has changed. By running the buildbots on a number of different systems we can ensure that `ns-3` builds and executes properly on all of its supported systems.

Users (and developers) typically will not interact with the buildbot system other than to read its messages regarding test results. If a failure is detected in one of the automated build and test jobs, the buildbot will send an email to the *ns-developers* mailing list. This email will look something like:

```
The Buildbot has detected a new failure of osx-ppc-g++-4.2 on NsNam.
Full details are available at:
  http://ns-regression.ee.washington.edu:8010/builders/osx-ppc-g%2B%2B-4.2/builds/0
```

```
Buildbot URL: http://ns-regression.ee.washington.edu:8010/
```

```
Buildslave for this Build: darwin-ppc
```

```
Build Reason: The web-page 'force build' button was pressed by 'ww': ww
```

```
Build Source Stamp: HEAD
Blamelist:
```

```
BUILD FAILED: failed shell_5 shell_6 shell_7 shell_8 shell_9 shell_10 shell_11 shell_12 s
```

```
sincerely,
-The Buildbot
```

In the full details URL shown in the email, one can search for the keyword `failed` and select the `stdio` link for the corresponding step to see the reason for the failure.

The buildbot will do its job quietly if there are no errors, and the system will undergo build and test cycles every day to verify that all is well.

3.2 Test.py

The buildbots use a Python program, `test.py`, that is responsible for running all of the tests and collecting the resulting reports into a human-readable form. This program is also available for use by users and developers as well.

`test.py` is very flexible in allowing the user to specify the number and kind of tests to run; and also the amount and kind of output to generate.

By default, `test.py` will run all available tests and report status back in a very concise form. Running the command,

```
./test.py
```

will result in a number of PASS, FAIL, CRASH or SKIP indications followed by the kind of test that was run and its display name.

```
Waf: Entering directory '/home/craigdo/repos/ns-3-allinone-test/ns-3-dev/build'
Waf: Leaving directory '/home/craigdo/repos/ns-3-allinone-test/ns-3-dev/build'
'build' finished successfully (0.939s)
FAIL: TestSuite ns3-wifi-propagation-loss-models
PASS: TestSuite object-name-service
PASS: TestSuite pcap-file-object
PASS: TestSuite ns3-tcp-cwnd
...

PASS: TestSuite ns3-tcp-interopability
PASS: Example csma-broadcast
PASS: Example csma-multicast
```

This mode is indented to be used by users who are interested in determining if their distribution is working correctly, and by developers who are interested in determining if changes they have made have caused any regressions.

If one specifies an optional output style, one can generate detailed descriptions of the tests and status. Available styles are `text` and `HTML`. The buildbots will select the `HTML` option to generate `HTML` test reports for the nightly builds using,

```
./test.py --html=nightly.html
```

In this case, an `HTML` file named “nightly.html” would be created with a pretty summary of the testing done. A “human readable” format is available for users interested in the details.

```
./test.py --text=results.txt
```

In the example above, the test suite checking the `ns-3` wireless device propagation loss models failed. By default no further information is provided.

To further explore the failure, `test.py` allows a single test suite to be specified. Running the command,

```
./test.py --suite=ns3-wifi-propagation-loss-models
```

results in that single test suite being run.

```
FAIL: TestSuite ns3-wifi-propagation-loss-models
```

To find detailed information regarding the failure, one must specify the kind of output desired. For example, most people will probably be interested in a text file:

```
./test.py --suite=ns3-wifi-propagation-loss-models --text=results.txt
```

This will result in that single test suite being run with the test status written to the file “results.txt”.

You should find something similar to the following in that file:

```
FAIL: Test Suite ‘‘ns3-wifi-propagation-loss-models’’ (real 0.02 user 0.01 system 0.00)
PASS: Test Case "Check ... Friis ... model ..." (real 0.01 user 0.00 system 0.00)
FAIL: Test Case "Check ... Log Distance ... model" (real 0.01 user 0.01 system 0.00)
Details:
```

```

Message:   Got unexpected SNR value
Condition: [long description of what actually failed]
Actual:    176.395
Limit:     176.407 +- 0.0005
File:      ../src/test/ns3wifi/propagation-loss-models-test-suite.cc
Line:      360

```

Notice that the Test Suite is composed of two Test Cases. The first test case checked the Friis propagation loss model and passed. The second test case failed checking the Log Distance propagation model. In this case, an SNR of 176.395 was found, and the test expected a value of 176.407 correct to three decimal places. The file which implemented the failing test is listed as well as the line of code which triggered the failure.

If you desire, you could just as easily have written an HTML file using the `--html` option as described above.

Typically a user will run all tests at least once after downloading `ns-3` to ensure that his or her environment has been built correctly and is generating correct results according to the test suites. Developers will typically run the test suites before and after making a change to ensure that they have not introduced a regression with their changes. In this case, developers may not want to run all tests, but only a subset. For example, the developer might only want to run the unit tests periodically while making changes to a repository. In this case, `test.py` can be told to constrain the types of tests being run to a particular class of tests. The following command will result in only the unit tests being run:

```
./test.py --constrain=unit
```

Similarly, the following command will result in only the example smoke tests being run:

```
./test.py --constrain=example
```

To see a quick list of the legal kinds of constraints, you can ask for them to be listed. The following command

```
./test.py --kinds
```

will result in the following list being displayed:

```

Waf: Entering directory '/home/craigdo/repos/ns-3-allinone-test/ns-3-dev/build'
Waf: Leaving directory '/home/craigdo/repos/ns-3-allinone-test/ns-3-dev/build'
'build' finished successfully (0.939s)Waf: Entering directory '/home/craigdo/repos/ns-3-a
bvt:          Build Verification Tests (to see if build completed successfully)
unit:         Unit Tests (within modules to check basic functionality)
system:       System Tests (spans modules to check integration of modules)
example:      Examples (to see if example programs run successfully)
performance: Performance Tests (check to see if the system is as fast as expected)

```

This list is displayed in increasing order of complexity of the tests. Any of these kinds of test can be provided as a constraint using the `--constraint` option.

To see a quick list of all of the test suites available, you can ask for them to be listed. The following command,

```
./test.py --list
```

will result in a list of the test suite being displayed, similar to :

```
Waf: Entering directory '/home/craigdo/repos/ns-3-allinone-test/ns-3-dev/build'
```

```

Waf: Leaving directory '/home/craigdo/repos/ns-3-allinone-test/ns-3-dev/build'
'build' finished successfully (0.939s)
ns3-wifi-propagation-loss-models
ns3-tcp-cwnd
ns3-tcp-interoperability
pcap-file-object
object-name-service
random-number-generators

```

Any of these listed suites can be selected to be run by itself using the `--suite` option as shown above.

Similarly to test suites, one can run a single example program using the `--example` option.

```

./test.py --example=udp-echo

```

results in that single example being run.

```
PASS: Example udp-echo
```

Normally when example programs are executed, they write a large amount of trace file data. This is normally saved to the base directory of the distribution (e.g., `/home/user/ns-3-dev`). When `test.py` runs an example, it really is completely unconcerned with the trace files. It just wants to determine if the example can be built and run without error. Since this is the case, the trace files are written into a `/tmp/unchecked-traces` directory. If you run the above example, you should be able to find the associated `udp-echo.tr` and `udp-echo-n-1.pcap` files there.

The list of available examples is defined by the contents of the “examples” directory in the distribution. If you select an example for execution using the `--example` option, `test.py` will not make any attempt to decide if the example has been configured or not, it will just try to run it and report the result of the attempt.

When `test.py` runs, by default it will first ensure that the system has been completely built. This can be defeated by selecting the `--nowaf` option.

```
./test.py --list --nowaf
```

will result in a list of the currently built test suites being displayed, similar to :

```

ns3-wifi-propagation-loss-models
ns3-tcp-cwnd
ns3-tcp-interoperability
pcap-file-object
object-name-service
random-number-generators

```

Note the absence of the Waf build messages.

Finally, `test.py` provides a `--verbose` option which will print large amounts of information about its progress. It is not expected that this will be terribly useful for most users.

3.3 Test Taxonomy

As mentioned above, tests are grouped into a number of broadly defined classifications to allow users to selectively run tests to address the different kinds of testing that need to be done.

- Build Verification Tests
- Unit Tests
- System Tests
- Examples
- Performance Tests

3.3.1 Build Verification Tests

These are relatively simple tests that are built along with the distribution and are used to make sure that the build is pretty much working. Our current unit tests live in the source files of the code they test and are built into the ns-3 modules; and so fit the description of BVTs. BVTs live in the same source code that is built into the ns-3 code. Our current tests are examples of this kind of test.

3.3.2 Unit Tests

Unit tests are more involved tests that go into detail to make sure that a piece of code works as advertised in isolation. There is really no reason for this kind of test to be built into an ns-3 module. It turns out, for example, that the unit tests for the object name service are about the same size as the object name service code itself. Unit tests are tests that check a single bit of functionality that are not built into the ns-3 code, but live in the same directory as the code it tests. It is possible that these tests check integration of multiple implementation files in a module as well. The file `src/core/names-test-suite.cc` is an example of this kind of test. The file `src/common/pcap-file-test-suite.cc` is another example that uses a known good pcap file as a test vector file. This file is stored locally in the `src/common` directory.

3.3.3 System Tests

System tests are those that involve more than one module in the system. We have lots of this kind of test running in our current regression framework, but they are overloaded examples. We provide a new place for this kind of test in the directory “`src/tests`”. The file `src/test/ns3tcp/ns3-interop-test-suite.cc` is an example of this kind of test. It uses NSC TCP to test the ns-3 TCP implementation. Often there will be test vectors required for this kind of test, and they are stored in the directory where the test lives. For example, `ns3tcp-interop-response-vectors.pcap` is a file consisting of a number of TCP headers that are used as the expected responses of the ns-3 TCP under test to a stimulus generated by the NSC TCP which is used as a “known good” implementation.

3.3.4 Examples

The examples are tested by the framework to make sure they built and will run. Nothing is checked, and currently the pcap files are just written off into `/tmp` to be discarded. If the examples run (don't crash) they pass this smoke test.

3.3.5 Performance Tests

Performance tests are those which exercise a particular part of the system and determine if the tests have executed to completion in a reasonable time.

3.4 Running Tests

Tests are typically run using the high level `test.py` program. They can also be run “manually” using a low level test-runner executable directly from `waf`.

3.5 Running Tests Under the Test Runner Executable

The test-runner is the bridge from generic Python code to `ns-3` code. It is written in C++ and uses the automatic test discovery process in the `ns-3` code to find and allow execution of all of the various tests.

Although it may not be used directly very often, it is good to understand how `test.py` actually runs the various tests.

In order to execute the test-runner, you run it like any other `ns-3` executable – using `waf`. To get a list of available options, you can type:

```
./waf --run "test-runner --help"
```

You should see something like the following:

```
Waf: Entering directory '/home/craigdo/repos/ns-3-allinone-test/ns-3-dev/build'
Waf: Leaving directory '/home/craigdo/repos/ns-3-allinone-test/ns-3-dev/build'
'build' finished successfully (0.353s)
--basedir=dir:          Set the base directory (where to find src) to 'dir'
--constrain=test-type: Constrain checks to test suites of type 'test-type'
--help:                Print this message
--kinds:               List all of the available kinds of tests
--list:                List all of the test suites (optionally constrained by test-type)
--out=file-name:       Set the test status output file to 'file-name'
--suite=suite-name:    Run the test suite named 'suite-name'
--verbose:             Turn on messages in the run test suites
```

There are a number of things available to you which will be familiar to you if you have looked at `test.py`. This should be expected since the test-runner is just an interface between `test.py` and `ns-3`. You may notice that example-related commands are missing here. That is because the examples are really not `ns-3` tests. `test.py` runs them as if they were to present a unified testing environment, but they are really completely different and not to be found here.

One new option that appears here is the `--basedir` option. It turns out that the tests may need to reference the source directory of the `ns-3` distribution to find local data, so a base directory is always required to run a test. To run one of the tests directly from the test-runner, you will need to specify the test suite to run along with the base directory. So you could do,

```
./waf --run "test-runner --basedir='pwd' --suite=pcap-file-object"
```

Note the “backward” quotation marks on the `pwd` command. This will run the `pcap-file-object` test quietly. The only indication that you will get that the test passed is the

absence of a message from `waf` saying that the program returned something other than a zero exit code. To get some output from the test, you need to specify an output file to which the tests will write their XML status using the `--out` option. You need to be careful interpreting the results because the test suites will *append* results onto this file. Try,

```
./waf --run "test-runner --basedir='pwd' --suite=pcap-file-object --out=myfile.xml'"
```

If you look at the file `myfile.xml` you should see something like,

```
<TestSuite>
  <SuiteName>pcap-file-object</SuiteName>
  <TestCase>
    <CaseName>Check to see that PcapFile::Open with mode 'w' works</CaseName>
    <CaseResult>PASS</CaseResult>
    <CaseTime>real 0.00 user 0.00 system 0.00</CaseTime>
  </TestCase>
  <TestCase>
    <CaseName>Check to see that PcapFile::Open with mode 'r' works</CaseName>
    <CaseResult>PASS</CaseResult>
    <CaseTime>real 0.00 user 0.00 system 0.00</CaseTime>
  </TestCase>
  <TestCase>
    <CaseName>Check to see that PcapFile::Open with mode 'a' works</CaseName>
    <CaseResult>PASS</CaseResult>
    <CaseTime>real 0.00 user 0.00 system 0.00</CaseTime>
  </TestCase>
  <TestCase>
    <CaseName>Check to see that PcapFileHeader is managed correctly</CaseName>
    <CaseResult>PASS</CaseResult>
    <CaseTime>real 0.00 user 0.00 system 0.00</CaseTime>
  </TestCase>
  <TestCase>
    <CaseName>Check to see that PcapRecordHeader is managed correctly</CaseName>
    <CaseResult>PASS</CaseResult>
    <CaseTime>real 0.00 user 0.00 system 0.00</CaseTime>
  </TestCase>
  <TestCase>
    <CaseName>Check to see that PcapFile can read out a known good pcap file</CaseName>
    <CaseResult>PASS</CaseResult>
    <CaseTime>real 0.00 user 0.00 system 0.00</CaseTime>
  </TestCase>
  <SuiteResult>PASS</SuiteResult>
  <SuiteTime>real 0.00 user 0.00 system 0.00</SuiteTime>
</TestSuite>
```

If you are familiar with XML this should be fairly self-explanatory. It is also not a complete XML file since test suites are designed to have their output appended to a master XML status file as described in the `test.py` section.

3.6 Class TestRunner

The executables that run dedicated test programs use a `TestRunner` class. This class provides for automatic test registration and listing, as well as a way to execute the individual tests. Individual test suites use C++ global constructors to add themselves to a collection of test suites managed by the test runner. The test runner is used to list all of the available tests and to select a test to be run. This is a quite simple class that provides three static methods to provide or Adding and Getting test suites to a collection of tests. See the doxygen for class `ns3::TestRunner` for details

3.7 Test Suite

All ns-3 tests are classified into Test Suites and Test Cases. A test suite is a collection of test cases that completely exercise a given kind of functionality. As described above, test suites can be classified as,

- Build Verification Tests
- Unit Tests
- System Tests
- Examples
- Performance Tests

This classification is exported from the `TestSuite` class. This class is quite simple, existing only as a place to export this type and to accumulate test cases. From a user perspective, in order to create a new `TestSuite` in the system one only has to define a new class that inherits from class `TestSuite` and perform these two duties.

The following code will define a new class that can be run by `test.py` as a “unit” test with the display name, “my-test-suite-name”.

```
class MySuite : public TestSuite
{
public:
    MyTestSuite ();
};

MyTestSuite::MyTestSuite ()
    : TestSuite ("my-test-suite-name", UNIT)
{
    AddTestCase (new MyTestCase);
}

MyTestSuite myTestSuite;
```

The base class takes care of all of the registration and reporting required to be a good citizen in the test framework.

3.8 Test Case

Individual tests are created using a `TestCase` class. Common models for the use of a test case include "one test case per feature", and "one test case per method." Mixtures of these models may be used.

In order to create a new test case in the system, all one has to do is to inherit from the `TestCase` base class, override the constructor to give the test case a name and override the `DoRun` method to run the test.

```
class MyTestCase : public TestCase
{
    MyTestCase ();
    virtual bool DoRun (void);
};

MyTestCase::MyTestCase ()
    : TestCase ("Check some bit of functionality")
{
}

bool
MyTestCase::DoRun (void)
{
    NS_TEST_ASSERT_MSG_EQ (true, true, "Some failure message");
    return GetErrorStatus ();
}
```

3.9 Utilities

There are a number of utilities of various kinds that are also part of the testing framework. Examples include a generalized pcap file useful for storing test vectors; a generic container useful for transient storage of test vectors during test execution; and tools for generating presentations based on validation and verification testing results.

4 How to write tests

To be completed.

5 Propagation Loss Models

This chapter describes validation of ns-3 propagation loss models.

5.1 FriisPropagationLossModel

5.1.1 Model reference

From source: [Wireless Communications-Principles and Practice](#)

Given equation:

$$Pr = Pt * Gt * Gr * \lambda^2 / ((4 * \pi)^2 * d^2 * L)$$

$$Pt = 10^{(17.0206/10)} / 10^3 = .05035702$$

$$Pr = .05035702 * .125^2 / ((4 * \pi)^2 * d * 1) = 4.98265e-6 / d^2$$

$$\text{bandwidth} = 2.2 * 10^7$$

$$m_noiseFigure = 5.01187$$

$$\text{noiseFloor} = ((\text{Thermal noise (K)} * \text{BOLTZMANN} * \text{bandwidth}) * m_noiseFigure)$$

$$\text{noiseFloor} = ((290 * 1.3803 * 10^{-23} * 2.2 * 10^7) * 5.01187) = 4.41361e-13W$$

no interference, so SNR = Pr/4.41361e-13W

Distance	:: Pr	:: SNR
100	4.98265e-10W	1128.93
500	1.99306e-11W	45.1571
1000	4.98265e-12W	11.2893
2000	1.24566e-12W	2.82232
3000	5.53628e-13W	1.25436
4000	3.11416e-13W	0.70558
5000	1.99306e-13W	0.451571
6000	1.38407e-13W	0.313591

5.1.2 Validation test

Test program available online at: <http://xxx.xxx.com>

Taken at default settings (packetSize = 1000, numPackets = 1, lambda = 0.125, 802.11b at 2.4GHz):

Distance	:: Pr	:: SNR
100	4.98265e-10W	1128.93
500	1.99306e-11W	45.1571
1000	4.98265e-12W	11.2893
2000	1.24566e-12W	2.82232
3000	5.53628e-13W	1.25436
4000	3.11416e-13W	0.70558
5000	1.99306e-13W	0.451571
6000	1.38407e-13W	0.313591
7000	1.01687e-13W	0.230393
8000	7.78539e-14W	0.176395

5.1.3 Discussion

As can be seen, the SNR outputted from the simulator, and the SNR computed from the source's equation are identical.

5.2 LogDistancePropagationLossModel

5.2.1 Model reference

From source: [Urban Propagation Measurements and Statistical Path Loss Model at 3.5 GHz](#)

Given equation:

$$PL\{\text{dBm}\} = PL(d_0) + 10*n*\log(d/d_0) + X_s$$

PL(1) from friis at 2.4GHz: 40.045997dBm

$$PL\{\text{dBm}\} = 10*\log(.050357/Pr) = 40.045997 + 10*n*\log(d) + X_g$$

$$Pr = .050357/(10^{((40.045997 + 10*n*\log(d) + X_g)/10)})$$

bandwidth = $2.2*10^7$

m_noiseFigure = 5.01187

no interference, so SNR = Pr/4.41361e-13W

taking Xg to be constant at 0 to match ns-3 output:

Distance	::	Pr	::	SNR
10		4.98265e-9		11289.3
20		6.22831e-10		1411.16
40		7.78539e-11		176.407
60		2.30678e-11		52.2652
80		9.73173e-12		22.0494
100		4.98265e-12		11.2893
200		6.22831e-13		1.41116
500		3.98612e-14		.090314
1000		4.98265e-15		.011289

5.2.2 Validation test

Test program available online at: <http://xxx.xxx.com>

Taken at default settings (packetSize = 1000, numPackets = 1, exponent = 3, reference loss = 46.6777, 802.11b at 2.4GHz)

Distance	::	Pr	::	snr
10		4.98471e-9		11293.9
20		6.23089e-10		1411.74
40		7.78861e-11		176.468
60		2.30774e-11		52.2868
80		9.72576e-12		22.0585
100		4.98471e-12		11.2939
200		6.23089e-13		1.41174
500		3.98777e-14		0.0903516
1000		4.98471e-15		0.0112939

5.2.3 Discussion

There is a ~.04% error between these results. I do not believe this is due to rounding, as the results taken from the equation from the source match exactly with the Friis results taken at one less power of ten. (Friis and LogDistance can be modeled by $P_t G_t G_r \lambda^2 / (4\pi)^2 d^n L$, where n is the exponent. n is 2 for Friis, and 3 for logDistance, which accounts for the power of ten. ie: Friis at 100m is equivalent to LogDistance at 10m.) Perhaps the ns-3 takes the random number into account despite not being listed in the source.

(Index is nonexistent)