Two simple tools for testing wireless communication modules in OMNeT++

Laura Marie Feeney
Communication Networks and Systems Lab
Swedish Institute of Computer Science
lmfeeney@sics.se

ABSTRACT

This code contribution abstract presents two simple OMNeT++ modules, `ScriptApplLayer` and `EnsembleApplLayer`, which can be used for systematic testing of modules that model the operation of wireless NIC's. The main purpose of this document is to emphasize the importance of reproducible unit and ensemble tests, by describing how these modules were used in practice to demonstrate issues in long-standing code.

1. INTRODUCTION

Simulation enables performance evaluations that would otherwise be logistically difficult, especially for wireless communication. However, simulation also poses considerable risks, because an error in the design and implementation of simulation code can make the results meaningless or misleading. Researchers who expect to produce results of relevance to industry must address this concern. From this perspective, reliability of results can be more important than extending functionality.

Testing is particularly important for modular open source software like OMNeT++, where a simulation may be built out of components from variety of sources. For this reason, module developers should not merely assert that software has been tested, but also provide reproducible unit and ensemble tests, along with code. This practice not only allows potential users to judge whether code has been adequately tested, but also form the basis for regression or comparison tests if the module is extended or used in another context.

This abstract presents two simple OMNeT++ modules, `ScriptApplLayer` and `EnsembleApplLayer`, which can be used for systematic testing of low level wireless communication protocols. The goal is not to discuss the details of the modules themselves (they are trivial). Instead, the goal is to highlight the value of such tools, by describing how they were used to find a bug in long-standing code and to check the internal consistency of statistics collection.

2. SCRIPTAPPLLAYER MODULE

As the name implies, `ScriptApplLayer` is an application layer module that generates outbound messages according to a schedule specified in a script file. It enables the user to efficiently create a set of unit tests for wireless NIC (MAC and signal processing) modules, where each test is a simple scenario with specific timing constraints. The `ScriptApplLayer` takes as a parameter the name of a text file, containing a time-ordered list of transmissions.

```
time    src-host-id   dest-host-id
1.0     1             0
1.435   2             0
```

Figure 1: Example script `script31`.

3. USING SCRIPTAPPLLAYER

The `ScriptApplLayer` is useful for creating unit tests, especially for MAC layer protocols. This can be done using `omnetpp.ini` to build up an automated test suite that generates log and result files for each test.

```
[Run 31]
output-vector-file                =  omnetpp_31.vec
sim.host[*].appl.scriptFile       =  "script31"
sim.numHosts                      =  3
sim.host[*].test-specific.param   =  value
```

Figure 2: `omnetpp.ini` file.

The manual checking of event logs and result files for each test to verify that the modules are behaving as expected is tedious and time-consuming, making it important to record validated output for future use. The verification task is surprisingly much easier if the protocol timing parameters are set to large, round values; it is much easier to check that events occur at multiples of e.g. 1 or .1 seconds. These long intervals also make it much easier to create scripts with sequences of interleaved events. In general, using `CmdEnv` and scripts (e.g. `sh` or `python`) is far more efficient than any visual environment.

The `ScriptApplLayer` was used in validating `ucsma`, a "universal" CSMA MAC layer module, which was combined with the SnrDecider module of the mobility-fw to create a NIC. As contributed code in OMNeT++ 3.x, this is part of a long established codebase. However the tests show an anomalous behavior in `SnrDecider`, whereby a radio can transmit and

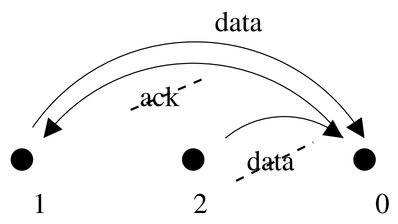

Figure 3: Test 31. Interfering transmission (CCA during Rx-Tx turnaround).

receive at the same time. (Note that this example should *not* be taken as any criticism of the developers' work; it is simply used to highlight the power of scripted scenarios for unit testing.)

Because the MAC layer has a relatively small number of operations and states, it is straightforward to list the interactions between hosts and systematically generate scripts to test them. Test 31 examined the case of a sender (host[2]) doing clear channel assessment (CCA) during the the rx-tx turnaround time of a receiver (host[0]) that is preparing to send an ACK (to another sender, host[1]).

In this test, we expect host[0] to record receiving a frame from host[1] and transmitting the ACK. But because host[2] does CCA for its pending frame during host[0]'s rx-tx turnaround and therefore begins transmitting its own frame, host[1] does not receive the ACK and records a failed transmission (retry limit set to zero). And because host[0] is transmitting the ACK during part of host[2]'s data transmission, it does not receive the data frame from host[2], which should record a failed transmission.

Yet in the `omnetpp.sca` file, we see:

```
scalar "sim.host[0].nic.mac" "dataForMe.samples" 2
scalar "sim.host[0].nic.mac" "controlOut.samples" 2
scalar "sim.host[1].nic.mgmt" "pktFail.samples" 1
scalar "sim.host[2].nic.mgmt" "pktFail.samples" 0
```

That is, host[0] seems to have received two frames and sent two ACKs: host[1] records a failure, because the ACK was lost, but host[2] does not. Examining the event logs shows that the SnrDecider accepted the frame because it did not check whether the radio was in the receive state for the entire frame duration.

Given this information, the bug was relatively easy to find and module `SnrDeciderFix` corrects the problem. It is also easy to check that the fix doesn't introduce new bugs by comparing output of all tests against previously validated output.

4. ENSEMBLEAPPLLAYER

The `EnsembleApplLayer` is an application layer module that supports ensemble tests, which allow the user to confirm that a protocol demonstrates predicted scaling behaviors.

Two modes of operation: "exponential" and "synchronized" are supported. For exponential traffic, each host generates packets with uniformly distributed destinations and exponentially distributed interarrival times. For synchronized traffic, every host but one (host[0]) periodically and simultaneously generates a packet, destined for host[0].

5. USING ENSEMBLEAPPLLAYER

The synchronized pattern of `EnsembleApplLayer` is useful for checking that MAC layer backoff behaves as expected, since it is easy to write an `omnetpp.ini` that defines a sequence of tests with an increasing number of competing senders. It can also be used for simple tests of e.g. saturated and interfering links.

The module is also useful for doing consistency checking on output statistics, as is seen from an example from testing of `ucsma`. It is possible to define a number of tautologies, expressing the properties of statistics summed over all hosts. For example:

`pktIn + pktDup - pktOut = controlOut - controlIn`.

Both sides of this equation express the number of lost ACKs. The *lhs* is the difference between the number of frames recorded as successfully received (including duplicates) and successfully transmitted. The *rhs* is the difference between the number of control frames recorded as sent and received.

6. IMPLICATIONS FOR OMNET++

Working with these testing tools has highlighted one area in which OMNeT++ might be extended to make testing easier: random number generators. The author hopes to raise these issues for further consideration.

It is easy to assign a fixed seed to each sequence of tests, since there is no need for true "randomness". But building unit tests would be easier if it were more straightforward to manage per-module RNG's, since each module's behavior would depend only on the scenario, rather than on the number of hosts using a shared sequence of RNG values.

For the unit tests above, the MAC layer was modified to (optionally) use a local RNG. The seed for each RNG was selected by trial and error, so that a pair of hosts had a brief sequence of identical backoff values. Since the unit tests consist of at most a few transmissions, doing this manually is feasible (but annoying).

In this context, it would also also be helpful to introduce a "non-RNG" plug-in, allowing a user to script the values returned by a particular module's RNG. This would considerably simplify creating tests requiring very specific backoff behavior, for example. Such a plugin is obviously rather fragile, but it would be used only in unit tests, where module behavior is being validated in detail.

7. CONCLUSION

This code contribution abstract has advocated unit and ensemble testing for OMNeT++ modules and described the use of two modules to support testing of low level communication protocols. The source code is available at `www.sics.se/nets/software`.