

Middleware Lab 03 :

Statistics collection and simulation campaigns

Guillaume-Jean Herbiet
<guillaume.herbiet@uni.lu>

December 19, 2008

Abstract

During the previous lab, you were given the description of an elaborated protocol for ad hoc services. Based on this information, you designed this protocol as a finite-state machine and implemented your design in OMNeT++ .

In this lab, you will focus on the statistics collection phase so as to be able to assess the protocol robustness under different conditions of mobility and configuration.

You will also learn how to easily perform an extensive simulation campaign with OMNeT++ so as to be able to get statistical confidence in your results and/or to assess your protocol performance under varying parameters.

1 Statistics collection and analysis methodology

Statistics collection and analysis is a crucial and difficult task. It is the final objective of network simulation and it should not be underestimated neither in time required, nor in complexity (as collecting statistics may require you to refine extensively your implementation).

For this phase to be efficient, it should be performed with some methodology: adding meaningless counters everywhere in your model and being overwhelmed by numbers to analyze is definitively not the good solution.

The following approach, while respected, may help you:

1. **Understand your design characteristics:** what are the main features of the protocol I have just designed? What are the functions I would like to assess? What are the protocol parameters I would like to vary so as to test what is the most suitable configuration?
2. **Find the proper statistics to collect:** based on the previous observations, what relevant metrics should I observe? Against what other metric/parameter would it be interesting to perform an analysis.
3. **Find how those statistics should be collected:** should those metrics be instantaneous values, simulation average, be collected independently on each host or be network global?
4. **Find how those statistics should be analyzed:** some metrics can be assessed against others (e.g. collisions vs. number of neighbors, throughput

vs. number of chat sessions). This can force you to collect additional statistics to serve as reference for analysis.

5. **Identify where in my model can each statistics be collected:** is it at application level (e.g. number of seen hosts), at MAC level (e.g. number of packet collisions), etc.
6. **Identify how on my model can each statistics be collected:** look at your implementation, at your network simulator API to find the most efficient/clean way to collect your statistics.
7. **Implement your statistics collection:** using the previous point. Don't forget to test on simple examples or by carefully debugging that correct values are collected.
8. **Consolidate your collected statistics:** to some point, your network simulator can perform this operation for you (see for instance histograms or the scalars and plove tools for OMNeT++). But you may also have to write some scripts for additional consolidation, or plotting custom graphs to represent your data. Perl and gnuplot are good candidates for this operation.
9. **Analyze your consolidated statistics** and conclude on your design performance.
10. **Iterate this process:** based on your conclusions, you may want to test with a different set of parameters or further investigate a given issue by collecting additional statistics on a given element (e.g. you found a lot of collisions and want to analyze on what packet kind do those collisions occur).

If you correctly follow this methodology, you should not be thinking about adding or modifying a statistic between steps 5 and 9 included.

2 Application to the ad hoc messaging protocol

2.1 Objectives

We will apply the previously-described methodology to the ad hoc service protocol we have defined and implemented in the previous labs.

Given a rough list of the statistics we want to focus on (i.e. steps 1 and 2 being performed), you will have to define how to actually collect the statistics, and implement the statistics collection.

2.2 List of statistics to collect

So as to assess the performance of our ad hoc messaging protocol, we will focus on four types of metrics:

- **Presence awareness related** metrics, to see if hosts have a correct image of the other users of the network;
- **State related** metrics, to analyze the behaviour of each node;

- **Call related** metrics, to study how efficiently are managed the different chat sessions;
- **Lower level** metrics, to have an idea of the messages drops and the associated reasons for losses.

2.2.1 Presence awareness metrics

Performance metrics

The basic service offered by our protocol is to provide each host a list of all the connected users inside the network. Therefore, it might be interesting to thoroughly examine the number of users seen by each host throughout the simulation and be able to provide an overall metric to assess the robustness of the presence awareness protocol.

Node centrality

It might be interesting to study the performance of presence awareness against a topological information. The network model we use is based on Wi-Fi which relies on a random-based scheme for channel access¹. Efficiency of this mechanism depends on the number of hosts contending.

To assess this, we will use a metric for the notion of **node centrality** which indicates to what extent can a node be considered as being in the center of the network or be a peripheral node.

There is many ways to measure this notion. In this lab, we will use an imperfect but simple metric which is the **coefficient of variation** of the hop count (i.e. number of relays) of all received messages by a node.

The coefficient of variation c_v is a normalized measure of dispersion of a probability distribution. It is defined as the ratio of the standard deviation σ to the mean μ , that is $c_v = \frac{\sigma}{\mu}$.

As in this lab we will not use mobility, the coefficient of variation could be computed using the average and standard deviation of the hop count during the whole simulation.

2.2.2 State related metrics

So as to have a better understanding of the ad hoc instant messaging protocol behavior, it is interesting to know the time spent by each node in each state.

Therefore, implement a computation of the simulation time spent in each state by each node of the network.

2.2.3 Call related metrics

Besides the presence awareness feature, the call management is the other important feature that directly impacts the user-perceived performance of our protocol. It is therefore vital to be able to assess how many sessions were successfully set up, correctly terminated or were avoided for any reason.

¹This is the CSMA/CA (Carrier Sensing Multiple Access with Collision Avoidance) mechanism

In the previous lab, the call session management protocol you have designed relies on messages exchanges for call initiation (it is a two-way handshake protocol) and termination. It also uses several timeouts to cancel ongoing calls or terminate call initiation process in case one or several messages are lost.

To provide useful metrics on this aspect, implement collection of the following statistics:

- number of total call attempts;
- number of call setup failures;
- number of successfully established calls;
- number of incorrect call termination;
- number of correct call termination
- number of timer expirations (separated metric for each timer type).

An efficient idea is to base the counting on the number of control messages sent/received. But remember that there might be different possibilities for a call session to avoid (refer to lab 02, figure 1 for examples of chat sequences).

As a consequence, you will probably need to verify some conditions not to count twice the same failure.

2.2.4 Lower-level metrics

It is also interesting to collect some metrics of lower-level (i.e. not directly related to the way the protocol works, but giving some information on its performance).

For this lab, we will focus on the implementation of three items:

- number of generated messages, which can help explaining some message loss, or give an idea of the resources needed for our protocol to perform;
- number of message collisions happening at the MAC layer, which can explain why some messages are lost;
- Number of maximum time-to-live (i.e. number of relay) reached. When this limit is reached, messages are dropped at the network layer to prevent them from looping indefinitely in the network.

3 Implementation advice

This section will help you implement the previously-described statistics.

Remember that in lab 01 section 6, you already learned how to subclass a module to add statistics collection and to visualize your results using `scalars` and `plove` tools. You can refer to this lab for a start (particularly for analyzing the hop count of messages).

3.1 Local statistics collection

Local statistics can be easily collected using `int`, `cLongHistogram` or `cOutVector` objects that you will update during the simulation inside a module and store inside the scalar or vector output file in the `finish()` method of that module.

As we used the finite-state-machine paradigm, the state transition function is a good place to perform some statistics collection, especially concerning the number of occurrence of a given state, or the time spent in each state.

3.2 Detecting collisions

As explained in paragraph 2.2.4, collisions between packets interfering with each other are detected at the MAC layer. As we use the 802.11 model built-in with the Mobility Framework in this lab, you will have to create a submodule of `Mac80211` (that I suggest you call `StatsMac80211`) to implement statistics collection of collisions.

In this new module, use the following code snippet to redefine the `handleLowerControl()` function. This function processes control messages that are sent up with arriving packets from the physical layer. The control messages contain information about the reception itself, and can tell you if the packet is correctly received or should be considered as collided.

Listing 1: Collecting statistics on collisions

```
void StatsMac80211::handleLowerControl(cMessage *m)
{
3   EV << "In handleLowerControl for statistics" << endl;
   // See the enum "ControlTypes_802_11" in "Consts80211.h"
   // for the possible values for the following switch
6   switch(m->kind())
   {
       case CONTROL_TYPE_1:
9       // ...
       case CONTROL_TYPE_N:
       // ...
12  }
   EV << "Ended handleLowerControl for statistics" << endl;
   Mac80211::handleLowerControl(m);
15 }
```

3.3 Global statistics collection

Collecting network-wide statistics requires an additional module to be created for sake of this only purpose. In this lab we will use an approach where each module which wants to record a global statistic will call a dedication of the global statistics collector module².

Therefore, you will have to implement a module called `StatsCollector`, which will be a submodule of the simulation and create a pointer to this statistics collector in each node module requiring to record a global statistics.

²This is the approach suggested in <http://www.omnetpp.org/pnwiki/index.php?n=Main.CollectingGlobalStatistics>

In the `/home/user/Development/templates/lab03` directory, you will find templates of a `.ned`, `.h` and `.cc` file that will help you design your statistics collector module.

Besides, the following code snippet can help you get a pointer to the `StatsCollector` module:

Listing 2: Getting a pointer to StatsCollector

```
// Create a pointer to a StatsCollector object
StatsCollector *collector;
3 // Use parentModule() enough to go up to
// the root simulation module
// then find its submodule called "statscollector"
6 cModule *m = parentModule()->parentModule()
    ->submodule("statscollector");
// Return value is a pointer to a cModule object,
9 // we need to safely cast it
// as a StatsCollector object, which is a child of cModule
collector = check_and_cast<StatsCollector*>(m);
```

A convenient thing to do is to name the global statistics the same as your local (i.e. per-node) statistics. In the output files (scalar and vector files) they will be differentiated by the module they were collected from.

What is also convenient with such a global module for statistics collection is that it will receive data from all the hosts. It is then easier to implement some intelligence not to record twice the same event (e.g. call failure) reported by two different nodes.

To prevent this, a possible approach is to record the call characteristics of the reported event (caller, callee) and the timestamp of the report, to store this in a round-robin array and to ignore additional reports with same call characteristics reported in a short period of time.

4 Efficient statistics collection and analysis

Now that you have implemented all your statistics collection functions, it is time to simulate your network behavior in order to collect data.

But, as many functions of your model use random numbers, it is necessary to launch a significant number of simulations and analyze results from all those different runs not to be victim of the phase effect³.

4.1 Different runs with different seeds

You may have noticed that any time you launched your simulation, you always obtained the same results. This is because the OMNeT++ random number generators were initiated with the same default seed (leading to the same sequence of random numbers). So as to gain statistical confidence in your results, you should run a significant number of simulations with different random seeds (20 is considered a minimum).

³An interesting reading on this topic is **On traffic phase effects in packet-switched gateways** available at <http://www-nrg.ee.lbl.gov/papers/phase.pdf>

OMNeT++ helps you with this task by using the concept of run. In the `omnetpp.ini` file, you can set up all your variable as usual. Besides that, you can define a set of numbered runs with different values for some parameters (including the seed) for each run.

In the `/home/user/Development/templates/lab03` you will find an example of a `omnetpp.ini` file with 20 different seeds set.

To launch a simulation with a given rune, you can use the `-r` option. To launch a sequence of runs, you can use a bash script. An example of such a script can be found in the `/home/user/Development/templates/lab03` directory or you can use the following code:

Listing 3: Bash script for multiple runs

```
#!/bin/bash
for i in `seq 1 20`;
3 do
    echo "run $i \"sim\""
    ./lab03 -r $i
6 done
```

4.2 Consolidating results

When you have performed a significant number of runs with varying seeds, you can analyze your results. OMNeT++ provides the `scalars` and `plove` tools to display histograms and vectors respectively.

But it might be interesting to display some metrics against others or to generate more elaborated plots. For this purpose, you can use Perl⁴ scripts to parse your results (scalar or vector file, or directly the complete output of your simulations) and generate gnuplot⁵ compatible data.

The `/home/user/Development/templates/lab03` directory contains Perl scripts (`.pl` files) and gnuplot command files (`.plt` files). You can use them and customize them to your needs.

Perl uses **regular expressions** while reading a file to find some text pattern. If a line matches a given pattern, it is analyzed and subsets of this line can be captured to Perl variables (e.g. `$1`, `$2`, etc.) by being put between parentheses and used for further processing (e.g. added to a `$sum` variable counting the sum of the values found in the file, etc.)

Would you like to test a regular expression against a given line of test, you can open the `/home/user/Regexp/regexp.html` file in a web browser and try it. It also contains some help on the regular expressions syntax.

Type `./mymetric.pl` to parse your results and generate a data file (`.dat`) that contains gnuplot-readable data. Then type `gnuplot mymetric.plt` to tell gnuplot to open the previously generated data file and display it. This will generate a graphic file representing the data.

⁴Perl is a scripting language which provides very convenient functions to open and parse text files to extract some data and perform computation on it

⁵gnuplot is an open source plotting software that works command line or by using a plot description file that tells the program how to display the data

5 Introduction of mobility

5.1 Mobility in network simulation

For results from a network simulation to be trustworthy, it is necessary for the simulation environment to be as close as possible to the reality. In particular, it is important for the model of users mobility to be carefully designed and to be meaningful (i.e. representing a plausible mobility pattern for the users in the considered use case).

The types of mobility models can be classified as follows:

- **traces models** that replay previously recorded traces in the simulation environment;
- **synthetic models** that try to reproduce realistic behavior of mobility without the use of traces. Synthetic models use most of the time a random or probability-based behavior. They are of two kinds:
 - **entity models** where each node movement is independent;
 - **group models** where nodes movements are correlated. Group models are often composed of a group component and a random additional individual motion.

The model we will implement is a synthetic entity model.

Would you be further interested by this topic, I recommend you the following readings:

- **A survey of mobility models for ad hoc networks**, available at http://www.cse.iitb.ac.in/~varsha/allpapers/wireless/adHocModels/mobility_models_survey.pdf
- **Random waypoint considered harmful**, at http://www.comsoc.org/confs/ieee-infocom/2003/papers/32_03.PDF

5.2 Constant Speed mobility

So as to assess the impact of mobility on our protocol performance, we will use the Constant Speed mobility model which is part of the Mobility Framework .

In this model, the user can define a velocity for each Host and an update interval. If the velocity is greater than zero (i.e. the Host is not stationary) the ConstSpeedMobility module calculates a random target position for the Host.

Depending to the update interval and the velocity it calculates the number of steps to reach the destination and the step-size. Every update interval ConstSpeedMobility calculates the new position on its way to the target position and updates the display. Once the target position is reached ConstSpeedMobility calculates a new target position.

5.3 Mobility configuration

So as to activate user mobility, change the speed variable in the Mobility Module section of the `mobility-fw.ini` file.

A good idea would be to set this parameter to different values and assess the importance of speed increase (what speed range is the protocol handling is a good question to answer).

5.4 Simulation campaign

Once you have ensured that your mobility model performs correctly, you can launch a real simulation campaign of at least 20 runs.

Analyze and consolidate your results to assess the impact of mobility on the protocol performance. It is part of your assignment to select meaningful statistics to study.

Conclusion

This series of lab introduced you with network simulators, an important tool in computer science research.

While playing with a modest application-layer protocol, you envisioned all the challenges related to the ad hoc nature of our network and were faced to the importance of finely modeling the application behaviour but also its environment (notably users mobility).

All those challenges are currently the object of many research work, as ad hoc network and associated services still remain a wide open area.