An Integrated Framework for Enabling Effective Data Collection and Statistical Analysis with ns-2

Claudio Cicconetti, Enzo Mingozzi, Giovanni Stea
Dipartimento di Ingegneria dell'Informazione
University of Pisa, Via Diotisalvi, 2 – 56100 Pisa, ITALY

{c.cicconetti, e.mingozzi, g.stea}@iet.unipi.it

ABSTRACT

The Network Simulator 2 (ns-2) is an open source tool for network simulation. When planning for large-scale simulation experiments, an efficient and flexible data collection and a statistically sound output data analysis are important aspects to keep in mind. Unfortunately, ns-2 offers little support for data collection, and statistical analysis of the simulation results is most often performed offline, using either home made code or available packages, which are not integrated with ns-2. In this paper we describe two complementary contributions: the first one consists of a set of C++ modules, that allow a flexible and efficient data collection; the second one is a software framework, which is fully integrated with ns-2, that performs all the operations required to carry out simulation experiments in a statistically sound way. Our framework allows a user to significantly reduce the post-processing overhead and to save simulation time, especially with large-scale simulations. Our code is publicly available at [3].

Categories and Subject Descriptors

G.3 [**Mathematics of Computing**]: Probability and Statistics – *statistical software*. I.6.7 [**Computing Methodologies**]: Simulation Support Systems – *environments*.

General Terms

Measurement, Performance, Experimentation, Verification.

Keywords

Simulation, ns-2, statistical analysis

1. INTRODUCTION

The *Network Simulator* (ns-2) [13] has established itself as a leader among a vast number of competitors (an impressive, though partial, list of which can be found in [17]). For instance, 43.8% of the simulations studies that appeared in the 2000-2005 proceedings of the ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc, [12]), were carried out using ns-2 [6]. Ns-2 is an open-source simulator, continuously enhanced and extended thanks to the contribution of a large community of researchers. Today, ns-2 includes a large number of network protocols, applications, algorithms, in varied environments, both wired and wireless, from large-scale Internet routing to wireless sensor networks.

WNS2'06, October 10, 2006, Pisa, Italy.

The only support for data collection in ns-2 is represented by *traces* and *monitors*. Traces record events related to the generation, enqueueing, forwarding, and dropping of packets. Each event corresponds to a line of ASCII characters, which contains information on the event type and the information stored into the packet. Such information includes the packet size, IPv4 source/destination addresses, TCP/UDP port numbers, and additional fields used for parsing purposes, such as flow/packet identifiers. In the case of wireless communications, further information is provided, including position and transmission energy of the device.

Monitors allow one to get statistical information regarding the behavior of one or more queues, possibly managing many flows simultaneously, both synthetically (i.e., per queue) and analytically (i.e., per flow). Monitors can trace the amount of arrived and departed packets and bytes, the amount of drops (both due to *early drop* strategies and to buffer overflow), and measure throughput. Furthermore, *flowmon* objects can monitor some basic statistics (i.e., average and variance) of delay.

However, traces and monitors alone are neither flexible nor efficient enough for collecting data in many practical cases. First, they only log events related to packet transmission. While packet-related events are often all that one needs to evaluate network performance from a user perspective (e.g. to compute the throughput or delay of traffic flows), evaluation of the internal state of network entities or protocols (e.g., computation of the size of the contention window of IEEE 802.11 devices, or the size of routing tables), requires a more general definition of the kind of events to be logged. Second, writing traces often adds a significant computational overhead, which slows down simulations considerably. Such overhead may become prohibitively large in *distributed* simulations, using for instance openmosix [9]. In such cases, in fact, ns-2 processes "migrate" from the home host (i.e. the host on which the ns-2 process has been run) towards a remote host, so as to balance the computational load among a cluster of workstations. However, traced events still have to be conveyed to the home host, which entails a high communication overhead, between the remote and the home host, in addition to the disk writes overhead. Third, traces and monitors log packet events on a *per-queue* basis (even though monitors can classify those events on a per-flow basis). In many cases, especially when running large-scale simulations, traffic is distinguished into *foreground* and *background* traffic [2]. The former, representing a small percentage of the overall traffic, is the one whose performance we actually want to measure, while the latter, representing the bulk of the traffic, is employed just to keep the network in the desired load condition. When such decoupling is in place, logging the events related to queues traversed by both foreground and background traffic just to extract the end to end delay of a tagged flow is clearly highly inefficient. The above mentioned problems require the ns-2 data collection subsystem to be suitably extended. More specifically, users need a *standard* way for i) specifying which events, not necessarily just packet-related ones, are to be logged, and ii) collecting data efficiently, i.e. so that the data collection itself does not slow simulations, and in such a way

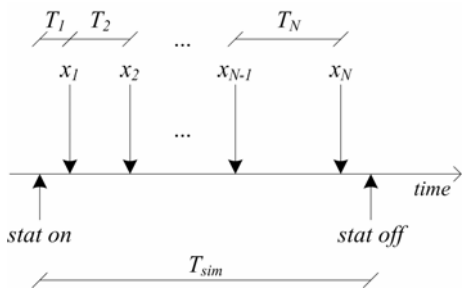

Figure 1. Metric types.

that the post-processing and parsing required to actually use those data for supporting performance evaluation (e.g., for plotting a graph) is kept reasonably low.

As regards statistical analysis of simulation output, many approaches can be applied [1]. Among these, the method of *independent replications* enjoys a number of desirable properties: it has good statistical performance if properly applied, it is easy to understand and implement, it applies to all types of output parameters, and, therefore, it is easy to be used for estimating different parameters for the same simulation model. This method consists of running a number of independent replications of the same simulation scenario, collecting one sample of each performance metric for each replication, and computing the sample mean of each metric with an associated confidence interval.

Ns-2 users willing to employ this method to get statistically sound results need to write their own code for i) generating independent replications of the same scenario, and ii) analyzing the output of the different replications so as to obtain averages and confidence intervals. Specialized tools for statistical data analysis, mostly commercial, are available, e.g. [11, 15, 18]: however, the high cost of some of them and their learning curve often discourage ns-2 users from actually employing them. Furthermore, they are not integrated with ns-2, so that some post-processing code for adapting the simulation output is still needed, as are scripts for generating and running independent replications. Writing this code is always time consuming, and, as shown in [6], does not necessarily lead to sound results. Thus, the ns-2 community would clearly benefit from having software modules that wrap the execution of ns-2 simulations, automating the generation of independent replications and the post-processing of data output according to statistically sound criteria.

In this paper, we propose two complementary contributions that overcome the above mentioned shortcomings. The first contribution is an extension of the data collection subsystem currently available in ns-2, which consists of a set of C++ classes to be integrated into the ns-2 distribution. More specifically, we devised a *Stat* class that receives samples of user-defined measures from *probes*, i.e. simple function calls inserted in C++ ns-2 modules. The *Stat* class, which handles an arbitrary amount of measures simultaneously, arranges the samples of a measure in a number of bins, from which probability distributions can easily be computed, and writes the outcome to a file. Furthermore, we derived two new connectors, namely *end to end tagger* (*e2et*) and *end to end monitor* (*e2em*), which sit in between a node and a sending (receiving) protocol agent. The e2et connector marks packets with a timestamp and a sequence number, so that the e2em connector is able to compute, and possibly send to the *Stat* class via probes, the end to end delay, inter-packet delay variation, loss rate and throughput of *selected flows*.

The second contribution is a software framework for data analysis and reporting, which is fully integrated with the last version of ns-2. This framework, consisting of two command-line programs (and a GUI that wraps them for better usability), allows a user to execute a number of independent replications of the same simulation scenario. Furthermore, it accepts as an input a text file listing a set of relevant measures, and it computes averages and confidence intervals of those measures. The text file is produced as an output by the *Stat* class. By using this framework, the ns-2 user is relieved of taking care of statistical aspects (such as writing code in the Tcl scenarios for selecting independent substreams of random numbers, or manually computing confidence intervals). Moreover, the software framework allows one to let simulation objectives guide the actual running of replications: for example, it is possible to specify that a new independent replication of a given scenario should be generated and run until the confidence interval of the required measure is below 10% of the sample mean.

The two contributions can be used together or independently, and only require non-intrusive modifications to the original ns-2 source code. Using the above mentioned framework makes simulative analysis with ns-2 easier and faster, and helps a user to preserve statistical soundness. The contributed code is publicly available at [3], as a patch for ns-2.29.

The rest of the paper is organized as follows. In Section II we describe the data collection subsystem. An introduction to the statistical analysis tools, both the GUI and the advanced command line tools, is provided in Section III. In Section IV we compare our integrated framework with the related work. Conclusions are drawn in Section V.

2. DATA COLLECTION SUBSYSTEM

The core of the contributed data collection subsystem is an ns-2 module, namely *Stat*, which collects raw samples from user-defined *probes*. Furthermore, we also developed ns-2 *connectors* which use *Stat* for helping users to easily collect typical end-to-end per-flow packet measures.

2.1 Stat Class

The basic mechanism provided for data collection is the static C++ class *Stat*. We assume that metrics of performance are described by random variables, whose stochastic characteristics are estimated from samples collected during the simulation runs. For the purpose of sample collection, we distinguish three types of metrics:

i. metrics which are intrinsically time averaged measures, e.g. the throughput or the loss rate;
ii. metrics described by a continuous-time stochastic process, e.g. the number of packets in a queue over time;
iii. metrics described by a discrete-time stochastic process, e.g. the end-to-end delay experienced by a flow of packets.

The above metrics are referred to into the source code as RATE, CONTINUOUS, and DISCRETE, respectively. For a given metric, data are collected for the purpose of estimating either the mean or the probability density function, with the exception of metrics of the first type, for which only the mean is meaningful.

More specifically, as far as data collection for mean estimation is concerned, one estimator of the mean per replication is computed in a different way, depending on the type of metric, as follows (see Fig. 1):

Table 1. TCL commands to the *Stat* class (`$ns` denotes the Tcl reference to the Simulator object).

`$ns stat file` *f*	the output file is *f*
`$ns stat on`	turn on samples collection
`$ns stat off`	turn off samples collection
`$ns stat print`	print the collected samples to *f*

$$y_{rate} = \frac{\sum_{i=1,\ldots,N} x_i}{T_{sim}}$$
$$y_{continuous} = \frac{\sum_{i=1,\ldots,N} x_i \cdot T_i}{\sum_{i=1,\ldots,N} T_i} \quad (1)$$
$$y_{discrete} = \frac{\sum_{i=1,\ldots,N} x_i}{N}$$

where x_i is the *i*-th sample collected from one replication run, T_{sim} is the replication run duration, N is the number of collected samples, and T_i is the duration of the estimation interval of the *i*-th sample. Estimates of mean are represented by a *double* number, irrespectively of the metric type.

Furthermore, the probability density function is estimated by partitioning the range of values of interest into a number of bins, and estimating the probability that the random variable belongs to each of them. The latter is equivalent to estimating the mean of as many new random variables as the number of bins, each one taking one (or T_i, in the case of continuous-time stochastic processes) as value, if the original random variable falls into the corresponding bin, and zero otherwise. Data structures to represent distribution estimates consist of an array of *double* numbers (bins) and the following *double* variables: the lower/upper bounds of the sampling interval, and the sampling size. When a sample *x* is inserted, *Stat* first checks whether *x* lies in the range [*lower bound*, *upper bound*]. If so, the value of bin *i* is incremented by one unit (or T_i) , where:

$$i = \left\lfloor \frac{x - lower\,bound}{bin\,size} \right\rfloor$$

The cumulative distribution function (c.d.f.) and probability mass function (p.m.f.) can be easily derived from this data structure. Two special bins, namely *underflow* (*UF*) and *overflow* (*OF*) store the number of samples which fall below the lower bound or above the upper bound. The content of these bins is then considered when computing the bin probability.

Regardless of the type of estimate, samples are collected through the invocation of the *put()* function, whose arguments are: the metric identifier, a numerical identifier, and the sample value to be added. The second argument is used to distinguish among different entities which might generate the same metric, e.g. throughput samples being added from several traffic flows. Since all the data structures and member functions of *Stat* are static, the programmer can call the *Stat::put()* function from any point in the ns-2 code[1], to insert a metric sample.

Metrics are configured in the file *common/statconf.h*, in which they are associated with a C++ enumerate identifier and a unique string name at compile time. Furthermore, the size, number and limits of bins are also defined for distribution measures.

[1] The line `#include <common/stat.h>` must be added to the source file.

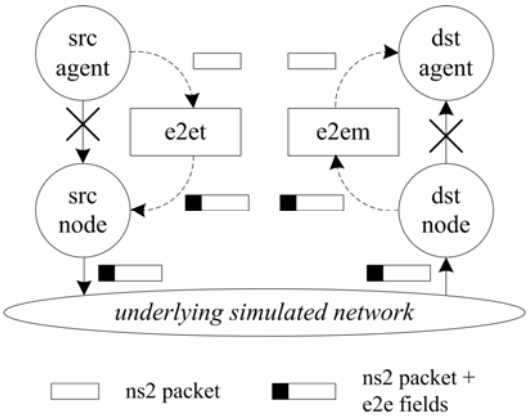

Figure 2. *e2et* and *e2em* insertion in ns2.

The Tcl interface for the *Stat* class is reported in Table 1. The `file` command sets the name of the output file, whose format is described in [3]. The `on` and `off` commands are used to (re)activate and deactivate, respectively, the collection of samples. For instance, in steady-state simulation, these commands can be used to avoid collecting samples during the warm up transient period. Finally, mean estimates and probability density functions are output to file *f* via the `print` command.

2.2 Per-flow Packet Measures

The above sample collection framework allows the maximum flexibility, since it allows a user to select his/her own measures in a standard way, and to insert probes at will in the code. However, a user is often concerned about end-to-end performance measures for a (small) subset of the traffic traversing its network, possibly a single flow. For this reason, we have developed two classes of *connectors* which allow a user to perform the task of collecting end-to-end measures efficiently, regardless of the traversed networks' types. The first one, named *end to end tagger* (*e2et*), marks packets with an absolute timestamp and a sequence number, which are added to the common packet header. The second one, named *end to end monitor* (*e2em*), can then compute the end-to-end delay and jitter based on the timestamp written by the e2et. Furthermore, it can detect packet losses by looking at the sequence number, and it can compute the throughput. Each e2em object adds samples to the *Stat* class through the *put()* function, thus tracking the following metric:

- *e2e_tpt*: throughput (bytes/s), average metric;
- *e2e_owpl*: one-way packet loss probability, average metric;
- *e2e_owd*: one-way delay (s), discrete-time distribution metric (by default, the sampling range is [0 s, 1 s], with 100 bins);
- *e2e_ipdv*: IP delay variation (s), discrete-time distribution metric (by default, the sampling range is [-0.5 s, 0.5 s], with 100 bins).

The two connectors can be seamlessly inserted between a protocol agent, which in fact allocates (or deallocates) packets, and the node the latter is attached to, as shown in Fig. 2. This is done via the following contributed wrapper functions:

```
$agtsrc attach-e2et $e2et
$agtdst attach-e2em $e2em
```

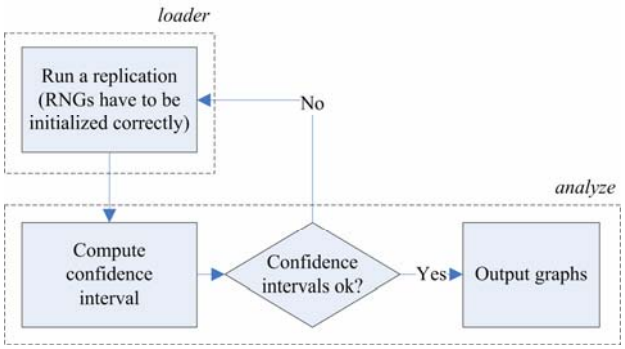

Figure 3. Framework for the analysis using the method of independent replications.

where, `$agtsrc` and `$agtdst` are the source and destination agents, respectively, and `$e2et` and `$e2em` are the end-to-end tagger and monitor objects, respectively.

In ns-2, agents are responsible for the fragmentation of packets, based on a user-specified Maximum Transfer Unit (MTU), called `packetSize_` in ns-2 terminology. Thus, *e2em* and *e2et* seamlessly integrate with any underlying network module which applies layer-2 fragmentation, provided that fragmented packets are reassembled before reaching the destination agent.

3. STATISTICAL ANALYSIS TOOLS

A statistically sound simulation analysis, according to the well-known method of independent replications, requires iterating the following steps:

- generate a new independent replication of a scenario;
- simulate the independent replication;
- collect the relevant measures.

The above steps are repeated until the relevant measures exhibit a satisfactory convergence: for instance, until the relative width of a confidence interval of the desired level is achieved, or until a maximum number of iterations has been performed. However, a user cannot know in advance the number of replications required to achieve the desired confidence. Therefore, users normally set a default number of replications to be executed, and control *a posteriori* whether the results have already reached the required level of confidence or more replications are needed. Thus, either a user directly interacts with the simulator, which considerably slows down the simulation process, or she specifies a "very large" number of independent replications, thus probably wasting simulation time. Furthermore, a simulative analysis almost always entails simulating a *set* of similar scenarios, which only differ in the value of one or few parameters. In these cases, it is hard to determine *a priori* a number of replications which is "large enough" for all scenarios. For instance, if the average delay at a node is to be evaluated as a function of the offered load, it is expectable that more and more replications will be needed in order to achieve a given relative width of the confidence interval as the offered load approaches the point of congestion.

This motivates the effort of writing a framework, illustrated in Fig. 3, that wraps the execution of single ns-2 simulation runs, automating the process of generation, execution and analysis of independent replications of a scenario. When using this framework, a user specifies the termination conditions in terms of a desired relative width of the confidence interval (i.e., "stop when the 95% confidence interval of the average delay is 10% of the sample mean"). Thus, the user needs not further interact with the simulator, while being assured

```
proc getopt {argc argv} {
  global opt
  for {set i 0} {$i < $argc} {incr i} {
    set arg [lindex $argv $i]
    if {[string range $arg 0 0] != "-"} \
      continue
    set name [string range $arg 1 end]
    set opt($name) \
        [lindex $argv [expr $i+1]]
  }
}
```

Figure 7. Tcl function to parse command-line arguments.

that the number of independent replications is always the minimum required to achieve its objectives.

The framework consists of two programs: *analyzer* and *loader*. The first one takes as an input a configuration file, in which the user defines the *analyzer* behavior, including:

- the minimum number of runs required, if any;
- the maximum number of runs required, if any;
- the set of relevant measures. A relevant averaged measure is identified by metric name and ID. A relevant distribution measure is identified by the metric name, ID, and performance index (i.e. p.m.f. or c.d.f.).

A comprehensive definition of the *analyzer*'s input file configuration parameters can be found in [3]. For each relevant measure, the user specifies:

- whether this measure is used by *analyze* to determine the termination condition; in this case, the confidence level and the relative width of the confidence interval are required;
- whether this measure should be plotted; in this case, the confidence level used for plotting is required.

At the end of each replication, *analyzer* reads the samples collected by the *Stat* class, stored into file *f* (i.e. the output filename specified via the `$ns stat file` *f*) and computes the confidence intervals as specified in its configuration file. If the termination condition is met by all the relevant measures or the maximum number of runs is reached, *analyzer* saves the output measures and terminates. Otherwise, it instructs the *loader* program to run a new replication.

On the other hand, the *loader* program executes the following two operations cyclically: (i) run a new replication of a simulation scenario; (ii) wait for an input from *analyzer*, and either go back to (i), if more replications are required, or exit. When *loader* runs a new replication, it initializes the RNGs so that the latter is independent from completed replications.

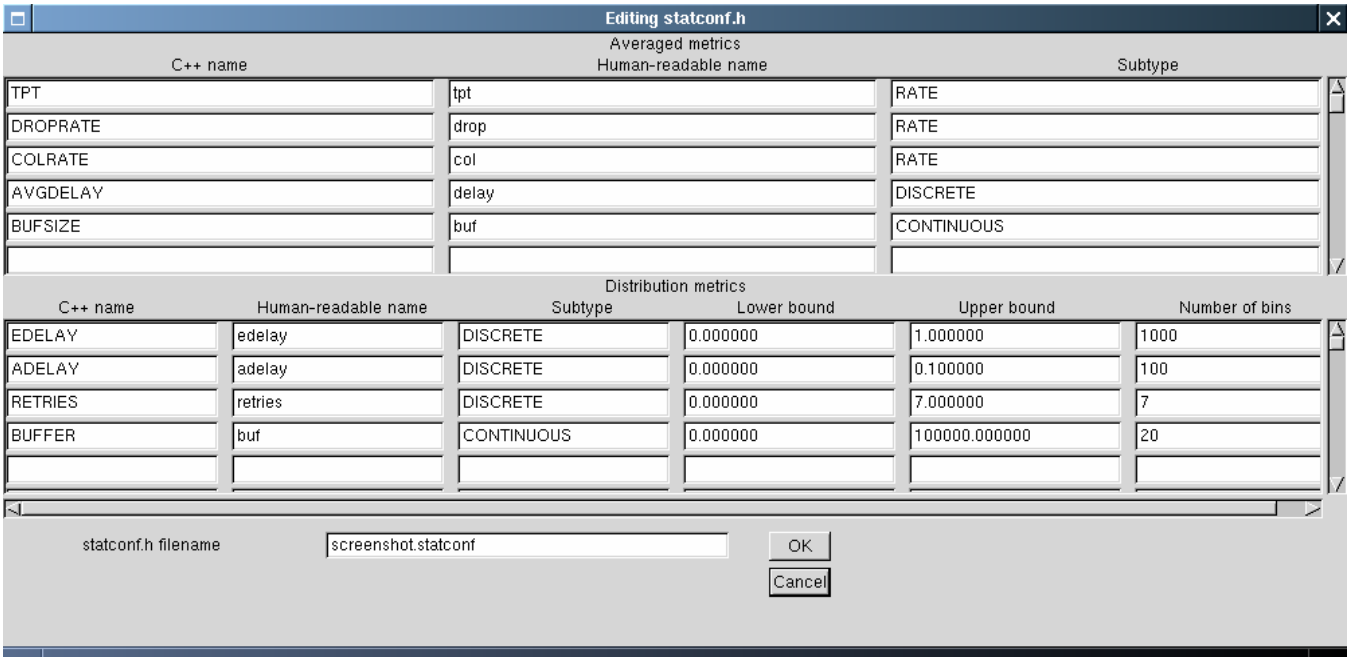

Figure 5. GUI: statconf.h editor.

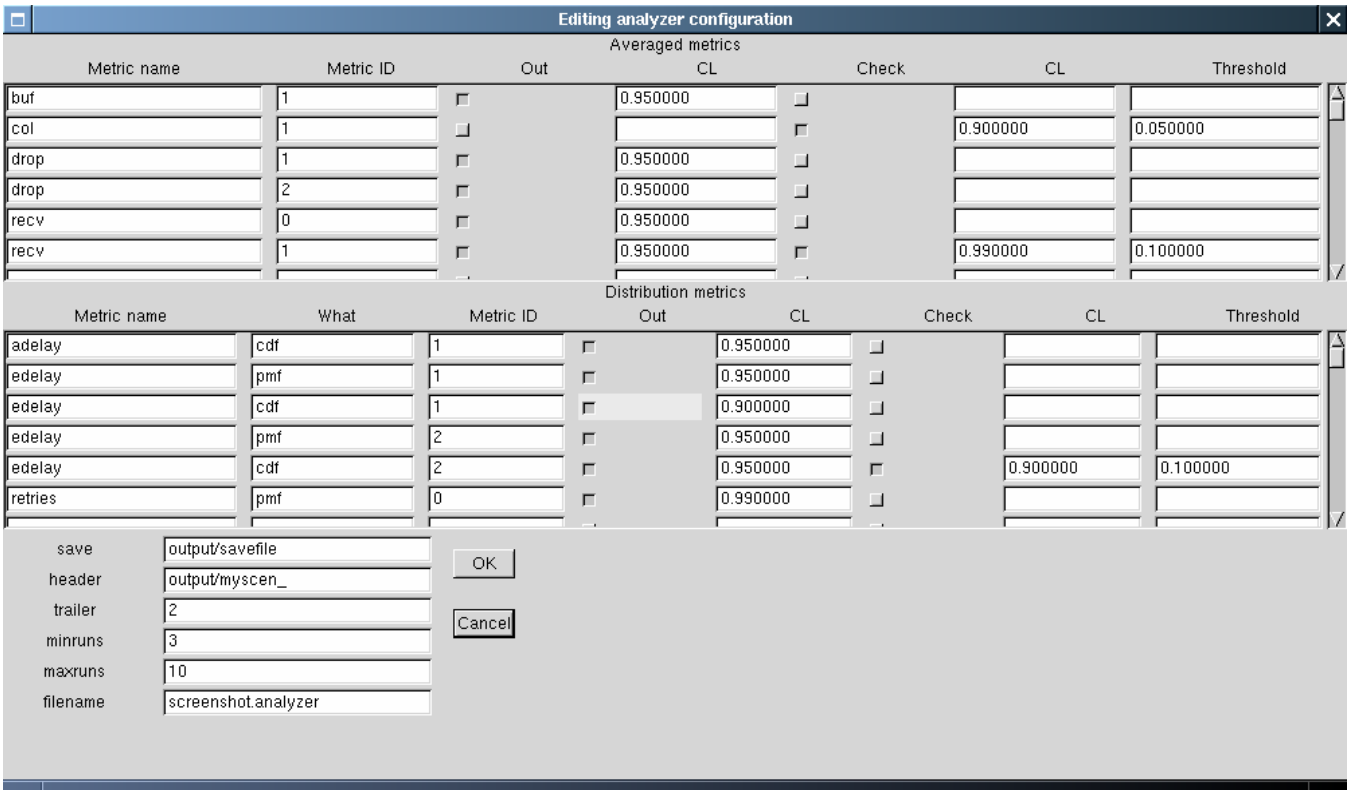

Figure 6. GUI: analyzer configuration editor.

To generate random numbers, ns-2 employs a combined multiple recursive generator proposed in [8], which provides many independent streams, each of which consists of several substreams. Therefore, the correct way to initialize RNGs with the method of independent replications in ns-2 is setting each RNG to a different substream for each replication. This process is left to the programmer by ns-2, which makes RNGs initialization error-prone. As part of our contribution, we then modified the RNG definition, so that independent seeds initialization is performed automatically. The user is only required to specify a different run identifier to distinguish among different replications of the same scenario, via the `run-identifier` Tcl command on the ns-2 simulator object. Note that the modified RNG is backward compatible, i.e. existing scripts in which generators are seeded manually will still work the same way.

4. GUI

The software described in this section can be used through command line. However, in order to make the whole framework more usable, we have developed a GUI, which serves two different purposes. Firstly, it generates the *common/statconf.h* file, which defines the relevant measures to be computed and is used to configure the *Stat* class. Secondly, it serves as a front-end for generating the *analyzer* configuration file. The GUI has been developed in C++ using the GTK+ libraries [4], which are free software and have been ported to several platforms.

In Fig. 5 we show the visual editor of the *Stat* configuration file (i.e. *common/statconf.h*). As can be seen, the first two columns of both averaged and distribution measures are used to define the C++ and human-readable measure names. The former is used in the source code as the first argument of the *Stat*::*put()* function, whereas the latter is used as part of the plot filenames. Additionally, averaged metrics require the metric subtype to be specified (i.e. one of RATE, CONTINUOUS, DISCRETE). Additionally, distribution measure require the lower and upper bounds, and the number of bins, on a per-measure basis.

In Fig. 6 we show the visual editor for the *analyzer* configuration file. The general parameters (e.g. minimum/maximum number of runs) can be configured in the bottom window section; averaged and distribution measures are configured in the top window section. Note that measures are identified using the human-readable identifier from the *statconf.h* file, not the C++ one. A row is meaningful if the metric name and ID are non-empty (and the performance index field, in the case of distribution metrics), and at least one of the 'out' (i.e. plot this measure) and 'check' (i.e. use this metric to check the termination condition) buttons are checked. Note that it is possible to specify a different confidence level for the 'out' and 'check' cases, if needed.

Finally, the GUI allows the user to specify a directory in which to save the files output by *analyzer*. Each file is named after the measure it contains, wrapped around by user-definable header and trailer. The saved files can be directly given as an input to graphic packages like *gnuplot* and *xgraph*, and we also provide shell scripts to convert them to other formats (e.g. the one required by *OriginPro* [15]).

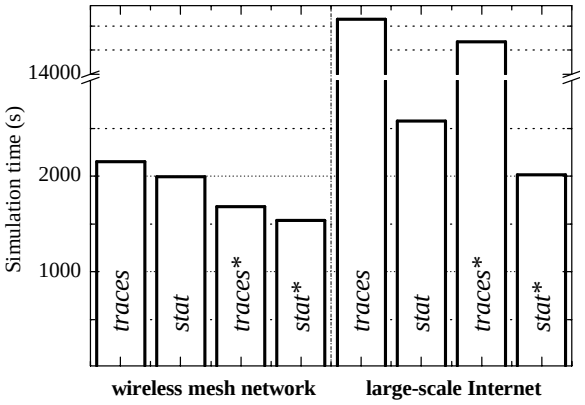

Figure 8. Simulation time of two different scenarios, using the contributed framework or ns2 traces, with (*) or without compilation optimization.

4.1 Comprehensive Example

In this section we summarize all the steps that are required to use the contributed framework from scratch. We assume that a UNIX-like environment is used, and that the command line interpreter is used. Firstly, download the *analyzer* and *loader* programs from [3] and compile them. Secondly, install the patch for ns-2.29 from [3], modify the file *common/statconf.h* to include the set of relevant metrics, add as many calls to the *Stat::put()* function as required by the simulation scenario, and compile ns-2. At this point, everything is compiled and ready to be used.

Assume that *example.tcl* is the Tcl file that contains the simulation scenario. First of all, the Tcl scenario has to be modified by inserting the procedure shown in Fig. 7, because *loader* assumes that the Tcl ns-2 scenario accepts the following command-line arguments: '-out' to set the filename for the communication between ns-2 and *analyzer*; '-run' to set the run identifier. Then, the following lines have to be added to the TCL ns-2 scenario in order to configure and activate the data collection subsystem:

```
$ns stat run-identifier $opt(run)
$ns stat file $opt(out)
$ns at warm "$ns stat on"
$ns at finish "$ns stat print"
```

where *warm* is the simulated time starting from which statistics have to be collected (if terminating simulation, *warm* = 0), and *finish* is the simulation duration. We stress the fact that all the random number generators used in simulation do *not* have to be initialized.

Furthermore, a user needs to write the *analyze* configuration file (say, *config*) to decide the set of measures that have to be printed, the confidence level of each measure, and the output filenames, which can be done through the GUI.

Finally, simulation and data analysis is started by issuing the following commands:

mkfifo in out
analyzer config out in
loader in out ns *example.tcl* args

where ‘ns’ is the path of the ns-2 executable, and ‘args’ is the list of arguments, if any, required by *example.tcl*.

5. RELATED WORK

There are many tools available for the analysis of ns-2 trace files. For instance, tracegraph [10] is a graphical tool for data presentation, written in Matlab [11], which parses ns-2 traces and produces the plots of a wide variety of performance measures. Many others can be found in the homepages of research groups using ns-2, or in the archive of the *ns-users* mailing list [13]. However, as already discussed, the use of traces may require a huge amount of disk space, which entails additional simulation time and high post-processing overhead.

To show the impact of writing ns-2 traces on the simulation time, we carried out two different simulation scenarios[2]. The first one consists of 25 wireless nodes using IEEE 802.11 at 2 Mb/s, arranged in a grid of 5×5 nodes, where each node has a greedy TCP connection with the middle node in the grid. The transmission range has been set to the grid step size, thus packets have to be relayed by intermediate nodes from the source to the destination. The second scenario represents a metropolitan value-added services distribution network. The latter is a tree network, in which pair of leaf nodes represent home users, which communicate with each other using VoIP and videoconference, and also watch either video-on-demand transmitted by the root node through the distribution network. The capacity of the links ranges from 80Gbps core links to 4Mbps access links. Foreground traffic is represented by *one* stream of each of the above mentioned types, whose end-to-end delay is to be measured. A suitable amount of background traffic is added to keep the links loaded up to a given extent. The Tcl scripts used to produce both scenarios are available in [3]. In Fig. 8 we report the simulation time of both scenarios using the *Stat* class and writing ns-2 traces, with and without compilation optimizations (i.e. ‘-O4’ and ‘-O0’ gcc options, respectively). In the first scenario, writing down traces yields an additional simulation time of 8% to 10%. In fact, since the average throughput of TCP connections is very low, due to the scarcity of wireless bandwidth, trace files are quite small. On the other hand, in the second scenario, traces are very large because of the high loads simulated, thus simulations last 6 to 7 times longer than without writing traces. Moreover, the overhead arising from the post-processing of traces is likely to be simply unfeasible.

Furthermore, there are some tools that collect the samples to be plotted during the simulation, thus eliminating the overhead of writing/analyzing traces. However, to the best of our knowledge, none of them is as general as our contribution, in terms of the types of measures that can be collected. Furthermore, output data analysis, such as confidence intervals computation, still has to be performed manually or using home-made tools. An example is the RPI graph and statistics package for ns-2 [5], which provides the user with a set of Tcl classes to generate several typical plots.

[2] The simulations were carried out using a machine with a single Intel Pentium IV 3.0 GHz CPU (512 KB L2 cache), 1 GB of main memory, and a hard disk with U-ATA66 controller. The Linux distribution was Slackware 10.0, with kernel 2.6.10.

Finally, we describe the work which is more closely related to our contribution: Akaroa [16]. The latter is a commercial product (free for teaching and non-profit research activities at universities) developed at the University of Canterbury in Christchurch, New Zealand, since 1992. Akaroa is a software framework to perform multiple replications in parallel on different processors. Each replication instance continuously sends observations of the relevant simulation parameters to a central process. The central process estimates the mean value of each parameter and, if the required simulation accuracy is reached, terminates the simulation. Even though Akaroa is a solid and stable framework to perform statistical analysis of simulation results in a pristine way, it was not explicitly designed for ns-2, which makes it difficult to integrate and use. The latest patch that allows ns-2 to be used in combination with Akaroa refers to version 2.26, and can be found in [14]. Furthermore, Akaroa provides the user with “low-level” statistical measures, which have to be manipulated adequately before presentation. On the other hand, our contribution provide ns-2 users with a simple way to define and use the measures that are relevant to their simulations, and produces final results automatically.

6. CONCLUSIONS

In this work we described a software framework for ns-2 to perform statistically sound simulations in a simple and efficient way. The framework consists of several parts. Firstly, the contributed C++ class, namely *Stat*, allows the user to add observation points into the ns-2 source code to collect samples of relevant measures. *Stat* is flexible enough to accommodate different types of measures, without the need of the user to modify the *Stat* implementation. Secondly, a pair of stand-alone tools, namely *analyzer* and *loader*, are used to run independent replications of the same simulation scenario until a specified level of accuracy is obtained. The estimated means and confidence intervals of the relevant measures are output by *analyzer* so that they can immediately be plotted using standard graph tools. Both *Stat* and *analyzer* are set up based on input configuration text files, which can be produced in a visual manner using a GTK+ front-end. Therefore, our framework reduces the burden of manually performing many, error-prone steps of network simulation with ns-2. Furthermore, it does so in an efficient manner, without requiring to write/process large amounts of data, as in the case of trace-based analysis.

7. ACKNOWLEDGMENTS

This work has been carried out within the framework of the QUASAR project, funded by the *Italian Ministry of Education, University and Research* (*MIUR*).

8. REFERENCES

[1] C. Alexopulos. A review of advanced methods for simulation output analysis. Proc. *Winter Simulation Conference 1994*, Lake Buena Vista, USA, Dec. 11-14, 1994, pp. 133-140.

[2] C. Cicconetti, M. L. Garcia-Osma, X. Masip, J. Sá Silva, G. Santoro, G. Stea, H. Tarasiuk. Simulation Model for End-to-end QoS across Heterogeneous Networks. Proc. IPS-MoMe 2005, Warsaw, Poland, Mar. 14-15, 2005.

[3] C. Cicconetti, E. Mingozzi, G. Stea. Measurement module for ns-2. http://info.iet.unipi.it/~cng/ns2measure/, last update May 2006.

[4] The Gimp Toolkit. http://www.gtk.org/, continuously updated.

[5] D. Harrisod. RPI ns-2 graphing and statistics package. http://networks.ecse.rpi.edu/~harrisod/graph.html, last update Dec. 2005.

[6] S. Kurkowski, T. Camp, M. Colagrosso. MANET simulation studies: the incredibles. Proc. *ACM SIGMOBILE MC^{2}R*, vol. 9, no. 4, Oct. 2005, pp. 50-61.

[7] A. M. Law, W. D. Kelton. Simulation modeling and analysis. Third edition, *McGraw-Hill*, 2000.

[8] P. L'Ecuyer. Good parameters and implementations for combined multiple recursive random number generators. *Operations Research*, vol. 47, no. 1, 1999, pp. 159-164.

[9] R. Lottaux, P. Gallard, G. Vallee, C. Morin, B. Boissinot. OpenMosix, OpenSSI and Kerrighed: a comparative study. Proc. *CCGrid 2005*, vol. 2, Cardiff, UK, 9-12 May 2005, pp. 1016-1023.

[10] J. Malek. Trace graph – Network Simulator NS-2 trace files analyser. http://www.tracegraph.com/, last update Jan. 2006.

[11] Mathworks Matlab. Statistics toolbox. http://www.mathworks.com/.

[12] http://www.sigmobile.org/mobihoc/

[13] http://nsnam.isi.edu/nsnam/, last update Feb. 2006.

[14] H. Woesner, E. Gillich, A. Köpke. The ns-2/akaroa-2 project. http://www-tkn.ee.tu-berlin.de/research/ns-2_akaroa-2/ns.html, last update Oct. 2004.

[15] OriginLab OriginPro. http://www.originlab.pro/.

[16] K. Pawlikowski, V. W. C. Yau, D. McNickle. Distributed stochastic discrete-event simulation in parallel time. Proc. *Winter Simulation Conference 1994*, Lake Buena Vista, USA, Dec. 11-14, 1994, pp. 723-730.

[17] A. E. Rizzoli. A collection of modelling and simulation resources on the Internet. http://www.idsia.ch/~andrea/sim-tools.html, last update Dec. 2005.

[18] The R project for statistical computing. http://www.r-project.org/, last update Dec. 2005.