Published in the proceedings of the 11th International Workshop on Computer-Aided Modeling, Analysis and Design of Communication Links and Networks, 8-9 June 2006, Trento, Italy, which should be cited to refer to this work.
DOI: https://doi.org/10.1109/CAMAD.2006.1649720

A Network Simulator for Education and Fast Protocol Development: Nessi

Jürgen Ehrensberger, Jérôme Vernez, Stephan Robert

***Abstract*— A new and simple Python network simulator Nessi is described in this paper. While other simulators focus on minimizing the simulation time, Nessi tries to minimize the development time and the difficulties to implement a new simulation model. As such, it is mainly oriented toward educational use, where it enables students to implement or modify simulation models of protocols with minimal overhead. A second application of Nessi is for verification and performance evaluation of new protocols, where it allows the developer to easily explore different options.**

Nessi comes with an easy to use graphical interface that allows the user to interactively monitor the behavior of a simulation, to modify simulation parameters and to plot results.

***Index Terms*—Network simulator, Python, protocol.**

I. Introduction

THE development of such a simulator can serve many purposes: First, it allows a student or a researcher to study mechanisms of existing protocols like CSMA (Carrier Sense Multiple Access) or more complex mechanisms (IEEE 802.3 or 802.11 for instance). Second, it allows one to study the behavior and the performances of complex networks. Finally, it may be used to develop new protocols and to evaluate their correctness and performances.

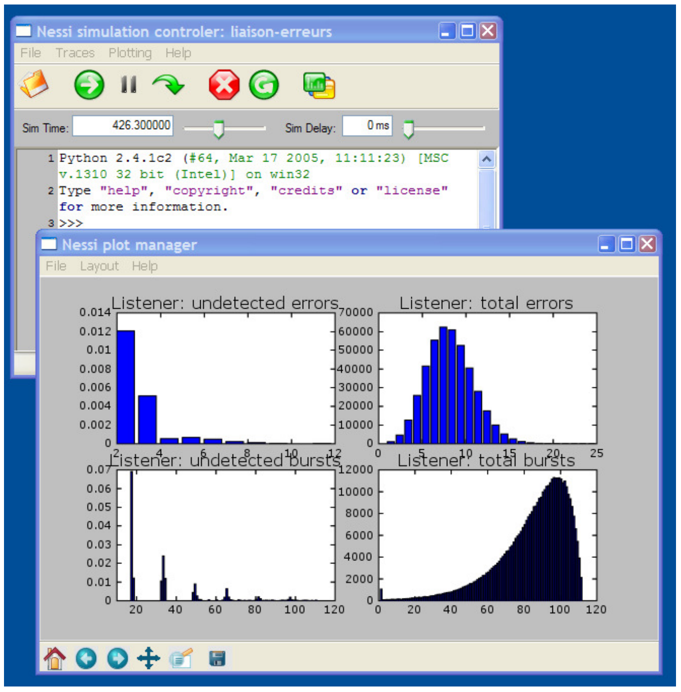
The Network Simulator ns-2 is the de-facto standard for network research in the scientific community. While being fast and offering a comprehensive set of simulation models, ns-2 has a steep learning curve and development of new protocols is difficult and error prone. Moreover, ns-2 is geared toward wireless / ad-hoc network research as well as network congestion control, covering only a part of the curriculum of telecommunications. Nessi may be seen as the complement of ns-2. It is much slower (by a factor of 20-100), but protocol development is much faster, easier to learn and less error prone. Nessi runs on Windows, Linux/Unix/BSD, MacOS and offers an easy-to-use graphical interface that allows students and researchers to interactively control simulations, to observe the behavior of the network and to produce graphical results.

J. Ehrensberger is a Professor at the University of Applied Sciences of Western Switzerland, HEIG-VD, 1400 Yverdon-les-Bains, Switzerland.

J. Vernez is a research assistant at the University of Applied Sciences of Western Switzerland, HEIG-VD, 1400 Yverdon-les-Bains,

S. Robert is a Professor at the University of Applied Sciences of Western Switzerland, HEIG-VD, 1400 Yverdon-les-Bains, Switzerland.

Figure 1 shows the simulation control window and a results window, running on Windows.


Nessi simulation controler: liaison-erreurs
File Traces Plotting Help
Nessi plot manager
File Layout Help
Listener: undetected errors
Listener: total errors
Listener: undetected bursts
Listener: total bursts

Sim Time: 426.300000 Sim Delay: 0 ms

```
Python 2.4.1c2 (#64, Mar 17 2005, 11:11:23) [MSC
v.1310 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license"
for more information.
>>>
```

Figure 1 - Nessi graphical user interface

Nessi has been designed to simplify and to speed-up the development of simulation models. As such it is mainly oriented toward educational purposes, where students can implement the most important mechanisms of current protocols with a minimum of overhead. A second application of Nessi is rapid prototyping of new protocols in order to verify the correct functioning of the protocol. We've developed Nessi primarily as a network simulator for education and rapid protocol development. To achieve these goals, Nessi is implemented in Python [2], an easy to learn interpreted language with a very comprehensive set of libraries, in particular for numeric and scientific computing [3]. Since Python is an interpreted ('scripting') language, this choice appears unusual. However, development and debugging of simulation models in C/C++ takes days or weeks, for some minutes or hours of actual simulation time. In our experience, development time in Python is reduced by a factor of 5-10, compared to C/C++ or Java, such that the total time spent for development and simulating of a new protocol may be considerably shortened. Moreover, development in Python is far less error prone, such that the quality of simulation models, which are notoriously difficult to validate, may be increased.

While University of Applied Sciences HEIG-VD during the past two years, it is still in an early development state. The simulation framework, including, the scheduler, random number generators, packet format creation, network entities, and plotting and scripting capabilities, is virtually completed. However, only a rather limited set of network protocols have been implemented in Nessi up to now, including:

- ARQ protocols like Stop-and-Go, Go-back-N under different bit error models.
- Error control methods like IP checksum, Hamming, Codes, polynomial codes and CRC.
- CSMA methods, like Aloha, CSMA/CA, CSMA/CD,
- A complete implementation of Ethernet on shared media.
- Wireless LAN IEEE 802.11.

The rest of this article is organized as follows. Section II presents the general structure of a network model used in Nessi. Sections III–VI provide details on packet format creation, traffic generation, the event scheduler and result tracing. Section VII shortly presents the graphical user interface. Finally, Section VIII concludes the article.

II. A SIMPLE STRUCTURE

In Nessi, a simulation is defined by a script that describes the structure of the simulation model and the simulation parameters like duration, and statistics to compute. A simulation script is a short and generally very simple Python program that uses objects from the simulation framework to compose a simulation model. Figure 2 shows a simple simulation model which may be used to evaluate of the efficiency of error detection codes.

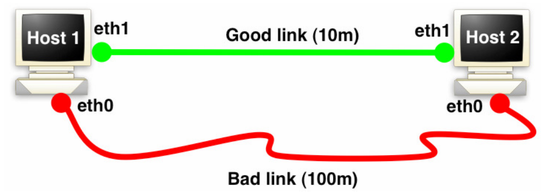


Figure 2 - Scheme of a simple point-to-point network. One link is error-free, the other simulates bit errors.

Figure 3 shows the simulation script for this network and Figure 4 illustrates the resulting hierarchy of network objects. Typically, a simulation script contains the following parts:

- Creation of the network elements (protocol entities, media, network interfaces, traffic sources and sinks).
- Building of the network topology by composing the network elements.
- Definition of statistics to measure.
- Definition of the simulation parameters (duration, etc.).

```
# Create the network
goodlink = PtPLink()
badlink = ErrorPtPLink()
badlink.errorModel('bernoulli', 1e-5)

# Create two nodes
hosts=[]
for i in range(2):
    h = Host()
    # On each host create interfaces eth0 and eth1
    for j in range(2):
        niu = NIC()
        h.addDevice(niu,"eth"+str(j))
        niu.addProtocol(FullDuplexPhy(), "phy")
        niu.addProtocol(PointToPointDL(), "dl")
    # eth0 is attached to the errored link,
    # eth1 to the error-free link
    h.eth0.attachToMedium(badlink)
    h.eth1.attachToMedium(goodlink)
    hosts.append(h)

h1 = h[0]
h2 = h[1]
# Connect the source and the sink
source = ChksumSource()
h1.addProtocol(source, "app")
source.registerLowerLayer(h1.eth0.dl)
source.registerLowerLayer(h1.eth1.dl)

sink = ChksumSink()
h2.addProtocol(sink, "app")
h2.eth0.dl.registerUpperLayer(sink)
h2.eth1.dl.registerUpperLayer(sink)

# Run the simulation
source.start()
RUN(1000)
```

Figure 3 - Simulation script that creates the network of Fig. 2

The types of network elements that are readily available are described in the following.

A. *Node*

A node is a container to which three types of network elements may be added: traffic sources and sinks, higher layer protocol entities and network devices. A node mainly provides these different elements with means to communicate with each other without via symbolic names. A protocol entity may for instance obtain the list of all installed network interfaces and decide over which interface is wants to send a data packet.

B. *Network Interface Units (NIU)*

A NIU is an interface between the physical transmission medium and higher layer protocols. NIUs follow the OSI reference model in as much as they allow the addition of a physical layer protocol (referred to as ‘phy’) and a data link layer protocol (‘dl’). A network interface can be attached to a transmission medium, such that the physical layer protocol entity can easily send and receive data over the medium.

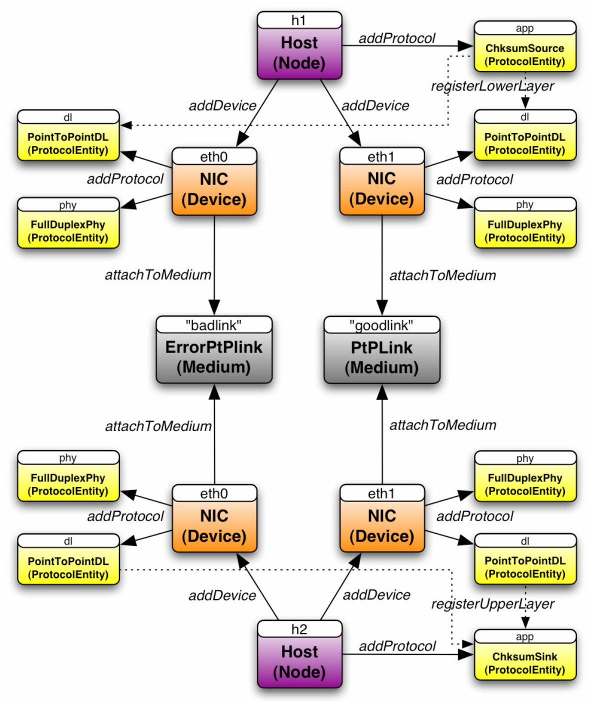


Figure 4 - Nessi structure corresponding to the scheme shown in Fig. 2. The name of each block corresponds to the name of the object which has been instantiated. The name below corresponds to its class and the name between parentheses is the parent class. The arrows correspond to methods calls which serve to link the different objects. The application layer entity of Host 1 is a traffic source that sends identical data packets over all available interfaces. The application layer entity of Host 2 is a data sink that receives all packets and compares them to test if the checksum correctly detects a bit error.

C. *Medium*

The medium simulates the transmission and propagation of data between all the network interfaces attached to it. The medium may be a point-to-point link, a shared bus or a radio channel. When an interface wants to attach to the medium, it gives its position (2- or 3-dimensional coordinates) in order to be able to calculate the propagation time.

Errors can be inserted to data to simulate an error channel. The following media are currently implemented:

- A shared bus, for simulating medium access protocols.
- An ideal point-to-point link, simulating propagation delays.
- A point-to-point link with bit errors with different error probabilities and distributions.
- A radio channel with bit errors.

D. *Protocol Entity*

A protocol entity implements a communication protocol to transmit data across the network. Following a layered protocol architecture, the protocols installed on a node exchange packets (‘protocol data units, PDUs) via “send” and “receive” calls. Physical and data link layer protocol entities are attached to network interfaces while higher layer protocols are directly attached to the host. Currently, the following protocols are implemented:

- Simple physical and data link layer protocols for point-to-point links.
- ARQ protocols link Stop-and-Go and Go-back-N over point-to-point links.
- Physical layer protocols for shared media like buses or radio channels.
- Medium access protocols for Aloha, CSMA, CSMA-CA.
- Ethernet (physical and MAC layer).
- Wireless LAN 802.11 (physical and MAC layer).
- Traffic generators as application layer protocols, which simulate the traffic of constant-bit-rate sources or self-similar sources (Web traffic).

III. A POWERFUL CLASS PDU GENERATOR

Nessi provides a very convenient method to create packet formats and to manipulate the content of data packets. The Ethernet packet format can be defined as shown in Figure 5.

```
PDUFormat = formatFactory(
    [('preamble', 'ByteField', 64),
     ('destAddr', 'MACAddr', 48, 'FF:FF:FF:FF:FF:FF'),
     ('srcAddr',  'MACAddr', 48, self.address),
     ('typeOrLength', 'Int', 16, 0x0800),
     ('data', 'ByteField', None, None),
     ('FCS', 'Int', 32, None)])
```

Figure 5 - Example of the creation of the packet format of Ethernet. Each line like preamble or FCS describes a packet field. A packet field is defined via a name, a data type, the length in bits and optionally a default value. Fields with a length of ‘None’ have variable length.

While it is not necessary for packet fields to be aligned on octet boundaries, the complete packet must have a length which is a multiple of a byte. The available data types are:

• ByteField:	An arbitrary sequence of bytes which is treated as a string. Example: 'abcd'.
• BitField:	Sequence of bits. The sequence may have an arbitrary length and is represented by a string of zeros and ones.
• MACAddr :	MAC address with a length of 48 bits Example : '20:C0:83:AD:33:01'.
• IPv4Addr:	IPv4 address, 32 bits, in dotted decimal notation. Example: '192.168.10.01'.
• Int :	An integer with an arbitrary length in bits.

The fields are defined as object attributes and directly accessible by their names. The following example shows how a new Ethernet packet is created and the destination MAC address is set to “00:11:22:AA:BB:CC”:

```
pdu = PDUFormat()
pdu.destAddr = “00:11:22:AA:BB:CC”
```

It has to be noted that the second instruction is not a simple assignment. Rather, a method is invoked (via the Python property mechanism) that checks if the assigned value is a correct MAC address and converts the address into the internal binary format.

IV. The Event Scheduler

Nessi is a discrete event simulator. The scheduler is the engine of the simulator that uses a virtual clock to execute operations at a scheduled time. The scheduler keeps an up-to-date list with the different events that have been scheduled. An event is defined by an execution time, a function to call and additional arguments. Nessi therefore implements and asynchronous programming model that does not allow for blocking function calls (calls that may block for an arbitrary time until the result is available). This differs from real-world protocol implementations that often use blocking calls. For instance, a protocol entity may call a function “receive” that only returns when a new packet is available. In Nessi, as in most discrete event simulators, this behavior has to be modeled via callback functions. The advantage of Nessi’s execution model is that it does not require threads, which considerably speeds up simulations and avoids problems due to non-reentrant functions.

V. Source and sink

A traffic source generates data packets according to a size and an interarrival distribution. As show in Figure 4, a protocol entity may be registered with the source via the function *registerLowerLayer*. Each time the source generates a new packet, it calls the ‘send’ methods of the registered protocol entities. The protocol entities then perform the actions to transmit the data across the network. At the receiver side, a traffic sink may be registered with a protocol entity via the function *registerUpperLayer*. When the protocol entity receives a data packet it passes it the sink which may simply discard the packet or perform more complicated operations. Currently, four types of traffic sources are implemented:

• CBRSource :	Generates data packets of fixed length, at fixed interarrival times.
• PoissonSource :	The packet size and the interarrival times have exponential distributions.
• WebSource :	This source simulate the traffic of http connections with self-similar behavior. It implements an empirical ON-OFF model described in [4] and [5]. The off periods have a length according to a Pareto distribution. The source transmits Web-pages with a size according to a log-normal distribution. During the transmission, packets have a constant size and are sent at constant rate. Self-similar behavior is created via the superposition of multiple connections.
• DLFlooder :	Sends fixed-size data packets as fast as the lower layers accept them.

VI. Statistics

An important element of every simulator is the computation of the statistics, which comprises two main tasks: measure one or more simulation parameter and compute the results based on the time-series on these parameters. Nessi provides different methods to measure simulations parameters:

- Explicit tracing: the simulation model contains instructions that write the current value of a parameter to a ‘trace collector’ each time the command is executed. The instructions to trace a parameter are therefore hard-coded into the simulation model.
- Sampling: a sampling function is executed in parallel which the simulation and samples the value of a parameter at different moments. Typically, a Poisson sampling is used, in which the intervals between the sampling moments are exponentially distributed. But other distributions (e.g., periodic sample with constant intervals) are possible.
- Variable tracing: the value of a variable is registered each time it is changes.

Most simulations use sampling to measure simulation parameters. The advantage is that the simulation model does not contain any instructions for result generation. They are added independently, e.g., in the simulation script. The same simulation model can therefore be used for different simulation experiments, without modifications.

Since Nessi is oriented toward education and the verification of protocol behavior, it provides support to observe and visualize the behavior of network protocols. This ‘activity tracing’ records the time and type of the actions of different protocol entities, which can be used to analyze and verify the coordination of the actions each system takes. To give a simple example, a sender can indicate the types of packets it sends, and a receiver can indicate the actions take upon the receipt of the different packets. This can be used the draw a sequence diagram of the operations of the network. While the tracing of activities is completely implemented, the visualization of the operations in the form of sequence diagrams or ‘arrow diagrams’ still has to be done.

In Nessi, all measured parameters are sent to a trace collector which can be configured by the simulation script. The trace collector can simply discard the values if not needed or redirect them to a file or to the graphical user interface, for interactive plotting, as shown in Figure 1.

The graphical user interface can be configured during the simulation run to add new statistics and to plot them as bar charts or line charts during the simulation. Moreover, the main window allows the user to change simulation parameters, like the traffic intensity, and observe immediately the effects, e,g., the evolution of lost packets. This is meant to help the student to gain a 'feeling' for the behavior of the network.

If the simulation parameters are written to a results file, they can be used to compute statistically valid results like mean values and confidence intervals. Python offers a very comprehensive library (see Scientific Python [3]) with mathematical operations in vector and matrix form, similar to Matlab™.

VII. GRAPHICAL INTERFACE

Nessi can either be executed as a batch program, for multiple simulation runs without user interaction, or as an interactive simulator with a graphical user interface. The main window of the simulator is shown in Figure 6. It allows the user:

- To load simulation scripts.
- To run, stop, pause and restart the simulation.
- To slow down the simulation for interactive control
- To interactively modify simulation parameters via a command line interface.
- To create graphics windows which interactively plot the statistics.

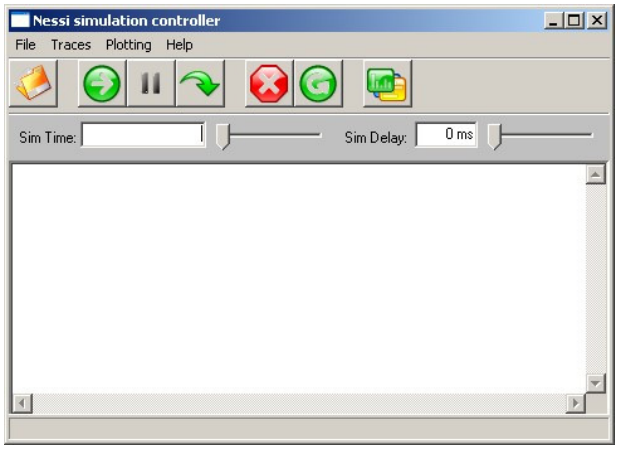


Figure 6 - Main interface of Nessi for interactive simulations

Both the core simulation of Nessi and the graphical user interface are portable. Nessi can thus be used under Windows, Linux/Unix and MacOS, with a look-and-feel similar to the actual platform.

Graphics windows allow the user to create sophisticated plots via a point-and-click interface. Currently, bar plots and line plot are implemented, as shown in Figure 1. The plots can be saved in different formats at any time.

VIII. CONCLUSION

We have presented Nessi, a Python network simulator for fast protocol development. Nessi is implemented in a scripting language Python which makes it slower that simulators implemted in C or even Java. However, the advantage of Nessi is that simulation models can be developed in a fraction of the time necessary with other simulations. Nessi therefore allows students to create or modify models of network protocols with minimal overhead and is thus perfectly suited for networking laboratories or semester projects. Another possible application of Nessi is to evaluate the performance and correctness of new protocols. Simulation models in Python are less error prone than implementations in C/C++ or Java. Moreover, even though simulation times with Nessi are quite long, protocol development is very fast, resulting in a total time for development and simulation that is shorter than with other simulators.

Nessi may be used as a batch simulator to perform multiple simulations runs which write statistics to trace files. However, Nessi excels as an interactive simulator that allows the user to start, stop, pause and slow-down simulation, to modify simulation parameters and to interactively plot the evolution of simulation results.

Currently only relatively basic protocols are implemented in Nessi, including ARQ protocols, CSMA protocols as well as Ethernet on shared media. We are currently working on the implementation or Wireless LAN 802.11, focusing especially on Quality of Service mechanisms defined in 802.11e.

REFERENCES

[1] Ns-2, The Network Simulator. http://www.isi.edu/nsnam/ns.
[2] The Python Programming Language. http://www.python.org.
[3] SciPy – Scientific Tools for Python. http://www.scipy.org.
[4] P. Barford and M. Crovella, 'Generating representative web workloads for network an server performance evaluation', Proc. 1998 ACM SIGMETRIC Intl. Conf. On Measurement and Modeling of Computer Systems, pp 151-160, July 1998.
[5] N. K. Shankaranarayanan, Zhimei Jian, and Partho Mishra, 'User-Perceived Performance of Web-browsing and Interactive Data in HFC Cable Access Networks.