



INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

## *Yet Another Network Simulator*

Mathieu Lacage — Tom Henderson

N° 5927

Juin 2006

Thème COM

 *Rapport  
de recherche*





## Yet Another Network Simulator

Mathieu Lacage , Tom Henderson

Thème COM — Systèmes communicants  
Projet planete

Rapport de recherche n° 5927 — Juin 2006 — 16 pages

**Abstract:** We report on the design objectives and initial design of a new discrete-event network simulator for the research community. Creating Yet Another Network Simulator (*yans*) is not the sort of prospect network researchers are happy to contemplate, but this effort may be timely given that *ns-2* is considering a major revision and is evaluating new simulator cores. We describe why we did not choose to build on existing tools such as *ns-2*, *GTNetS*, and OPNET, outline our functional requirements, provide a high-level view of the architecture and core components, and describe a new IEEE 802.11 model provided with *yans*.

**Key-words:** network simulation, discrete-event, emulation, 802.11

## Yet Another Network Simulator

**Résumé :** Nous présentons les objectifs et la conception initiale d'un nouveau simulateur réseau événementiel à temps-discret destiné à la communauté de recherche en réseaux. Le démarrage du projet ns-3 qui a commencé à évaluer de nouveaux cœurs de simulations constitue le moment idéal pour contribuer à l'architecture des outils que nous utiliserons demain: nous décrivons ainsi pourquoi nous n'avons pas choisi de ré-utiliser des outils existants tels que *ns-2*, *GTNetS* ou OPNET, présentons nos objectifs fonctionnels puis mettons en évidence leur impact sur l'architecture et les composants centraux de *yans*. Enfin, nous décrivons un nouveau modèle pour les réseaux IEEE 802.11 qui est intégré dans *yans*.

**Mots-clés :** simulation réseau, temps-discret, simulateur événementiel, emulation, 802.11

## 1 Introduction

This paper reports on the design and goals of a new discrete-event network simulator for Internet research. The title of the simulator (Yet Another Network Simulator, or *yans*) and of this paper explicitly begs the question of why, with a number of available existing network simulators to choose from, we would undertake to start over. This paper explains the rationale for *yans*, comparison to existing tools, and current and future design plans.

Our work on *yans* is an outgrowth of the INRIA Planete' research group's work on implementing an IEEE 802.11a/e MAC model for the *ns-2* simulator [1]. The impact of *ns-2* on networking research has been considerable. Brief surveys of the literature turn up large numbers of papers in most networking journals and conferences that cite usage of *ns-2*. It has arguably the largest model set for research on Internet protocols, and the source code is licensed (GNU GPLv2) appropriately for our project.

However, our initial work on *ns-2* revealed some limitations for our use:

- *coupling between various models is very high*: Many unrelated components, orthogonal features, and models depend on each other, sometimes in non-obvious ways. This often makes it impossible to combine various models together. For example, if one were to implement a new type of network node (a subclass of the Node class), it would be impossible to reuse the default implementations of the DSDV or DSR routing protocols because these depend on the MobileNode class.
- *object-oriented techniques have been widely ignored*: A lot of OTcl code, as well as some C++ code, tests for the type of the object manipulated before using it, rather than delegating the work to object-specific methods. For example, this makes it much harder to add new types of wireless routing protocols, requiring a careful audit of the code base to sprinkle the wireless code with yet another set of if/then/else statements testing for the type of routing protocol. Related to this, the C++ facilities for type-casting of pointers has been largely ignored. The integration of OTcl/C++ object bindings may have driven the design away from certain aspects of the C++ language.
- *the use of C++ standard library has been deprecated*: For historical reasons (compiler support), the use of C++ STL and constructs therein such as templates has been avoided by *ns-2*. However, compiler support for the standard library is now much improved since the time when the core *ns-2* architecture was defined.
- *coupling between C++ and OTcl is very high*: The use of the `otcl` and `tccl` libraries has encouraged the authors of models to split functionality between OTcl and C++ and make the C++ side of the model aware of the OTcl side and vice-versa. This sort of technique probably looks attractive from an abstract point of view, but its drawback is that it requires maintainers to spend their time trying to figure out where a given functionality is implemented (in OTcl or C++) and muddies the definition of an object's interface. Typically, it quickly becomes very hard to figure out what methods can be invoked on a given C++ object because part of its functionality is exported as C++ methods and the other part as OTcl methods. Of course, this problem is amplified when inheritance enters the game since both the OTcl and the

Table 1: Simulator licensing terms

|                  |   |
|------------------|---|
| GloMoSim/Qualnet | Academic until 2000 and commercial since then.  |
| <i>GTNetS</i>    | BSD-like with export restrictions for rtikit.   |
| OMNET++          | Academic and commercial for the core and diverse for the models.  |
| OPNET            | Academic and commercial: the academic version is limited in features.   |
| JiST/SWANS       | Restricted to academic use.   |
| SSFNet           | The main implementation of the SSF specification provided by Renesys is restricted to commercial use, or to academics only within the US. |

C++ methods of the parents are inherited. Experience has shown that debugging in this environment can be a challenge.

Besides *ns-2*, a number of other open-source simulators have been developed, including GloMoSim [2], NCTUns [3], *GTNetS* [4] (including the RTIKit library [5]), OMNET++ [6], SSFNet [7], and JiST [8]. Two popular commercial tools are OPNET [9] and QualNet [10].

We reviewed the licensing terms of these simulators and found that the restrictions were not acceptable for our project; all of the above are not completely freely and openly available or place restrictions on use of the software. What we understood of these various licenses is summarized in Table 1.

In light of the fact that several *ns-2* developers are interested in exploring a move to a new simulation core, we decided to experiment with our *yans* approach. The remainder of this paper describes the high-level goals and requirements of *yans*, the basic design, the IEEE 802.11a model, and reviews some of the performance characteristics of the resulting architecture.

## 2 *yans* goals and requirements

### 2.1 Licensing

We desire to develop our software with a well-known license that allows unencumbered research and use of the resulting simulator. Clearly, the GNU GPL or a BSD-style license

would fit the bill. However, because we would like to make sure every user contribute back his or her modifications to the simulator (whether these modifications are aimed at research or commercial use), we chose the GPLv2 without requesting any copyright assignment; i.e., each contributor owns the copyright of his contribution.

## 2.2 Architecture

While the licensing issues surrounding every open source project seem tractable, designing a software architecture which will be as successful in terms of number of users as *ns-2* and will be able to withstand this success is much harder: ensuring the long-term (15 to 20 years) architectural integrity of a reasonably-large codebase on which many contributors will work is very hard. The following paragraphs outline how we plan to learn from previous projects such as *ns-2*.

Clearly, the first lesson we learned from *ns-2* is to avoid a dual-language simulator<sup>1</sup> to decrease the overall complexity of the system. While it should be possible to use the simulator from any language (Python, Perl, tcl, java, etc.), writing new models for the simulator should be done in a single language. This is the architecture adopted by a lot of software projects: wrappers for the single-language application core are created as needed for each language of interest and numerous tools such as SWIG [11] have been designed to make this easier.

The single-language core was written in C++, mostly because we felt that the integration of existing C/C++ models would be easier.

## 2.3 Processes

Using a single language is a necessary first step to try to minimize the complexity of the simulator. However, it is far from being enough: it is also necessary to define clear contribution rules and processes to be able to deal with the decentralized development of models by multiple users in different countries, with different backgrounds. These rules must also be enforced during integration: the *ns-2* manual describes a simple coding style but large amounts of its code do not follow it.

*yans* thus defines:

- a coding style enforced during integration through code reviews: the coding style is described in the CONTRIBUTING file included in the *yans* distribution;
- an informal code review process: the authors who would like to see their code integrated in *yans* need to have their code reviewed at least once; and
- requirements on model maintainership: each model should have at least one maintainer. Models that lack maintainers will be removed from the *yans* source tree. The list of maintainers is described in the MAINTAINERS file included in the *yans* distribution.

---

<sup>1</sup>Before working on *yans*, we attempted to modify *ns-2* to be able to perform C++-only simulations but this exercise in refactoring was too difficult for us to complete it.

### 3 Functional requirements

*yans* was built with the idea that we wanted to make it very easy to perform a number of tasks which are often regarded as very hard and sometimes impossible with *ns-2* or other simulators:

1. Emulation: it should be easy to plug the simulator into a real network and make the simulated nodes exchange data with the real nodes.
2. Integration of user-space networking applications/daemons: it should be possible to re-compile some of the classic Unix daemons and make them use the simulator as their source of input/output.
3. Integration of kernel-space networking stacks: it should be possible to re-compile the networking stacks of open source operating systems in the simulator with a minimal amount of modifications.
4. Tracing: easy tracing/dumping of packets and interesting events, from deep in the networking stack, in widely-accepted formats (e.g., `pcap` for packet traces).
5. Scripting: the availability of at least one scripting language that wraps the simulation core and makes it possible to control most aspects of a simulation

As described below, our prototype has accomplished items 4, and 5 to date, and we believe we have paved the way for items 1, 2 and 3.

### 4 Architectural overview

*yans* is built around a C/C++ simulation core that provides:

- a simulation event scheduler (located in `src/simulator`), and
- a number of utility APIs used to implement various network models (located in `src/common`).

The rest of the C/C++ code implements models for various network components. *yans* also provides a default Python wrapper for the simulation core and the models bundled with it. This Python wrapper imposes negligible performance penalty on Python simulations compared to pure C++ simulations.

#### 4.1 The event scheduler

The event Scheduler provides a classic event scheduling API that allows its users to lookup the current simulation time, schedule events at arbitrary points in the future and cancel any event which has not yet expired.

However, some of its features are noteworthy. Its users can:

- Schedule events for the "end of simulation" time to release any resource acquired (typically, memory).
- Schedule events for the "now" time; that is, events that will be scheduled after the current event completes and before any other event runs. This feature can be used to avoid recursion and re-entrancy problems.

- Choose the underlying priority queue algorithm used to order events.
- Record to a text file the exact list of scheduling operations (such as inserts, deletes, etc.) and replay from a text file these events. This is especially useful to evaluate the performance of various scheduling algorithms on a given load.
- Use C++ templates to generate the code of forwarding events; these automatically-generated events can forward event notifications to arbitrary class methods or functions with an arbitrary number of per-event arguments.

The simulation time is maintained internally by a 64-bit integer in units of microseconds. While the Scheduler also exports this simulation time as a floating-point number in units of seconds, users are advised not to use it: past experience with *ns-2* models based on floating-point arithmetic for time has shown numerous problems related to accuracy. Indeed, most model developers assume that floating-point operations are performed with infinite arithmetic precision and ignore all the issues related to accuracy control. This makes it hard to ensure the reproduction of simulation scenarios and results across a large range of hardware and software platforms and sometimes leads to very hard to debug problems on certain platforms.

The scheduling order of events scheduled to expire at the same time is specified to be that of the insertion time. i.e.: the events inserted first are scheduled first. This order holds whatever the scheduling algorithm chosen: it is implemented by using an event sequence number, incremented for each insert.

Because it is possible to change the priority-queue algorithm used by the event scheduler, the complexity performance of the insert and the remove operations are not specified. However, the currently-implemented algorithms provide:

- Linked List:  $O(n)$  insert, and  $O(1)$  remove;
- Binary Heap:  $O(\log(n))$  insert and remove (*ns-2* is  $O(\log(n))$ ), insert,  $O(n)$  random remove and  $O(\log(n))$  first remove); and
- stdC++ map:  $O(\log(n))$  insert and remove.

## 4.2 Network model facilities

To make the implementation of network-related models as easy as possible, *yans* also provides a few very important facilities:

- callback objects that implement a C++ template-based version of the Functor design pattern,
- a packet API to create and manipulate packets,
- trace support classes, and
- a uniform random number generator based on the mrg32k3a random number generator used in *ns-2* and described in [12].

The callback API is absolutely fundamental to *yans*: its purpose is to minimize the overall coupling between various pieces of *yans* by making each module depend on the callback API itself rather than depend on other modules. It acts as a sort of third-party to which work is delegated and which forwards this work to the proper target module. This

callback API, being based on C++ templates, is type-safe; that is, it performs static type checks to enforce proper signature compatibility between callers and callees. The API is minimal, providing only two services:

- callback type declaration: a way to declare a type of callback with a given signature, and,
- callback instantiation: a way to instantiate a template-generated forwarding callback that can forward any calls to another C++ class member method or C++ function.

This callback API is already used extensively in most models currently available in *yans*. For example, our LLC/SNAP encapsulation module was made independent of the IPv4 and Arp layers with callbacks invoked whenever a payload must be forwarded to the higher layers.

The packet-manipulation API is based on the *skb/mbuf* APIs from the BSD/Linux operating systems: a packet is a buffer of bytes and it is possible to efficiently add and remove chunks of bytes from the start and the end of the buffer. Access to the underlying byte buffer is done only through a specialized Buffer API that provides convenient methods to serialize and deserialize packet headers and trailers to/from host and network format. It is also possible to efficiently add and remove an arbitrary number of tags to each packet to allow easy implementation of cross-layer mechanisms (for example, a 802.11e application could tag each packet with its 802.11e class to allow the MAC to accurately schedule packets according to their scheduling class and the negotiated scheduling policy).

This API should encourage model developers to make their simulation packets accurately reflect the exact structure of the simulated network packets. This should make it easy to exchange packets with real networks in realtime. Furthermore, generating conformant *pcap* traces out of such packets is trivial and generally improves the user experience.

The trace support is a bit less mature but it allows us already to provide two types of tracing:

- packet logging: whenever the model logs a packet, a user-provided callback is invoked.
- variable change logging: whenever the value of a variable changes (when being assigned to for example), a user-provided callback is invoked.

This framework was designed to offer a lot of flexibility: the users can select which events they want to monitor and they are free to use arbitrarily complex logic to decide when to log the events to a trace file. For convenience, we also provide a few default trace writers to simplify the task of serializing the events to a trace file in *pcap* format which can then be read back with third-party GUIs such as *Ethereal*.

The uniform number generator was included after extensive discussions with a few users: initially, we were planning to make *yans* depend on a third-party library such as *GSL* [13] which would have provided both such simple random number generation services but also more elaborate mathematical functions. However, the cost of increasing the size of the dependency list for *yans* and thus the risk of making *yans* harder to build and use for our users was considered too high; offering a built-in random number generator makes porting and using *yans* on numerous platforms easier. This also makes it much easier to ensure proper reproducibility of the simulation results, whatever the platform considered.

### 4.3 The default Simulation Models

*yans* comes with a small but focused set of default simulation models:

- a Thread model allows our users to mix classic sequential synchronous code with their event-driven asynchronous code;
- a Node model which implements a TCP/UDP/IPv4 stack and offers a socket-like API. The TCP stack is a port of the BSD4.4Lite TCP stack: it is distributed separately because its BSD license is not compatible with the GPL license of the simulator. Nodes are connected to each other through their Network Interfaces;
- an Ethernet Network Interface model; and
- a 802.11 Network Interface model which implements the IEEE 802.11 MAC DCF and the IEEE 802.11e EDCA and HCCA. It also implements a multirate PHY model based on piecewise SNIR calculations with an energy threshold.

## 5 Real-world code integration

Our interest in being able to integrate real-world code in the simulator comes from two very different perspectives:

- our limited development resources make us very sensitive to the possibility of reusing existing code, and
- we would like to simulate accurately a real-world network with its real-world bugs and limitations.

Doing so poses a number of very interesting technical challenges which we have spent considerable time to attempt to tackle:

1. the need to provide a process-driven API in an event-driven simulator,
2. the need to provide separate address spaces to separate simulated processes: global static variables must not be shared across simulated processes.
3. the need to provide a suitable build and link environment to both user-space and kernel-space code.

The thread model provided by *yans* offers a process-driven API (where address spaces are shared) that is built on top of the existing event-driven services. It uses a small user-space thread library that has been ported to x86 Linux and ppc32 OS X systems. Porting it further should not pose new problems and is a matter of manpower.

On systems based on ELF [14], our plan to deal with static variables in process context is to build the process code as a separate shared library with Position Independent Code. Then, *yans* would need to load the shared library itself in memory at runtime, once for each simulated process. The idea is that each simulated process would run the exact same code but mapped at different virtual addresses. On any modern operating system, this would consume only the address space and no real physical memory would be wasted, provided that the memory mappings are not writable and the code is not changed. Since each copy of the code of the simulated process accesses its static variables through the Global Offset Table whose position is relative to that of the code, each simulated process would access a

different copy of the GOT, which would achieve the desired effect, that is, the ability to use simulated-process-specific static variables.

Other systems which use other binary standards to deal with process-specific static variables could see similar solutions but our initial focus is on Linux ELF systems.

Providing a proper build and link environment to kernel-space code requires a careful re-implementation of the OS-specific libraries used by this kernel-space code. One such important library is the `skb` and the `mbuf` buffer abstractions of the Linux and the BSD kernels respectively. The Packet API used in our network models has been designed to make it possible to write a thin relatively simple wrapper around it to make it look like either an `skb` or an `mbuf`.

User-space code has similar requirements: it needs a socket and a `libc` implementation. The TCP and UDP models implemented in *yans* were designed to make such an implementation possible.

## 6 The 802.11 model

The core node-based models in *yans* allow the users to plug any number of network interfaces in each node and makes the network interface models completely independent of the other *yans* models. The 802.11 model was developed independently from *yans* itself (it was originally written for *ns-2*). It implements a MAC and a PHY layer that conform to the 802.11a specification. Work on full 802.11e support (including EDCA and HCCA) is underway at the time of this writing and should be complete by the end of August 2006.

### 6.1 The PHY model

Since we are unaware of any published detailed description of a complete wireless link model, this section summarizes the description of the BER calculations found in [15], presents the equations required to take into account the Forward Error Correction present in 802.11a, and describes the algorithm we implemented to decide whether or not a packet can be successfully received.

The PHY layer can be in one of three states:

- TX: the PHY is currently transmitting a signal on behalf of its associated MAC
- RX: the PHY is synchronized on a signal and is waiting until it has received its last bit to forward it to the MAC.
- IDLE: the PHY is not in the TX or RX states.

When the first bit of a new packet is received while the PHY is not IDLE (that is, it is already synchronized on the reception of another earlier packet or it is sending data itself), the received packet is dropped. Otherwise, if the PHY is IDLE, we calculate the received energy of the first bit of this new signal and compare it against our Energy Detection threshold (as defined by the Clear Channel Assessment function mode 1). If the energy of the packet  $k$  is higher, then the PHY moves to RX state and schedules an event when the

last bit of the packet is expected to be received. Otherwise, the PHY stays in IDLE state and drops the packet.

The energy of the received signal  $S(k, t)$  is assumed to be zero outside of the reception interval of packet  $k$  and is calculated from the transmission power with a path-loss propagation model in the reception interval:

$$P_l(d) = P_l(d_0) + n10\log_{10}\left(\frac{d}{d_0}\right) \quad (1)$$

where the path loss exponent,  $n$ , is chosen equal to 3, the reference distance,  $d_0$  is chosen equal to 1.0m and the reference energy  $P_l(d_0)$  is based based on a Friis propagation model:

$$P_l(d_0) = \frac{P_t G_t G_r \lambda^2}{16\pi^2 d_0^2 L} \quad (2)$$

where  $P_t$  represents the transmission power,  $G_t$ , the transmission gain (set to 1 dbm by default),  $G_r$  the reception gain (set to 1 dbm by default),  $\lambda$  the carrier wavelength,  $d_0 = 1$  and  $L$  is the system loss (chosen equal to 1 in our simulations).

When the last bit of the packet upon which the PHY is synchronized is received, we calculate the probability that the packet is received with any error,  $P_{err}(k)$ , to decide whether or not this packet could be successfully received or not: a random number  $rand$  is drawn from a uniform distribution and is compared against  $P_{err}(k)$ . If  $rand$  is larger than  $P_{err}(k)$ , then the packet is assumed to be successfully received. Otherwise, it is reported as an erroneous reception.

To evaluate  $P_{err}(k)$ , we start from the piecewise linear functions shown in figure 1 and calculate the Signal to Noise Interference Ratio function  $SNIR(k, t)$  with equation 3.

$$SNIR(k, t) = \frac{S_k(t)}{N_i(k, t) + N_f} \quad (3)$$

where  $N_f$  represents the noise floor which is a characteristic constant of the receiver circuitry and  $N_i(k, t)$  represents the interference noise, that is, the sum of the energy of all the other signals received on the same channel:

$$N_i(k, t) = \sum_{m \neq k} S(m, t) \quad (4)$$

From the  $SNIR(k, t)$  function, we can derive  $BER(k, t)$  for BPSK (see equation 5) and QAM (see equations 6, 7, and 8) modulations.

$$BER(k, t) = \frac{1}{2} \operatorname{erfc}\left(\sqrt{\frac{E_b}{N_0}}(k, t)\right) \quad (5)$$

$$BER(k, t) = 1 - (1 - P_{\sqrt{M}}(k, t))^2 \quad (6)$$

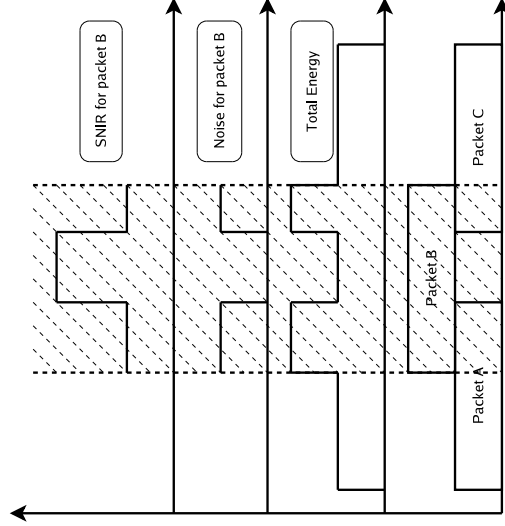


Figure 1: SNIR function over time

$$P_{\sqrt{M}}(k, t) = \left(1 - \frac{1}{\sqrt{M}}\right) \cdot X(k, t) \quad (7)$$

$$X(k, t) = \text{erfc} \left( \sqrt{\frac{1.5}{M-1} \cdot \log_2 M \cdot \frac{E_b}{N_0}(k, t)} \right) \quad (8)$$

where  $E_b$  is the energy per bit,  $N_0$  the noise power density, and  $\frac{E_b}{N_0}(k, t)$  is defined as:

$$\frac{E_b}{N_0}(k, t) = \text{SNIR}(k, t) \cdot \frac{B_t}{R_b(k, t)} \quad (9)$$

$B_t$  is the unspread bandwidth of the signal (that is, 20MHz for 802.11a) and  $R_b(k, t)$  is the bit-rate of the transmission mode used by signal  $k$  at time  $t$ .

Then, for each interval  $l$  where  $BER(k, t)$  and  $R_b(k, t)$  are constant, we define the  $P_e(k, l)$  function which represents an upper bound on the probability that an error is present in the chunk of bits located in interval  $l$  for packet  $k$ . If we assume an AWGN channel, binary convolutional coding (which is the case in 802.11a) and hard-decision Viterbi decoding,  $P_e(k, l)$  can be defined by the equations 10, 11 and 12 as detailed in [16].

$$P_e(k, l) \leq 1 - (1 - P_u(k, l))^{8 \cdot L(k, l)} \quad (10)$$

when  $L(k, l)$  is the size of the interval  $l$  in bits, and the union bound  $P_u(k, l)$  of the first-event error probability is given by:

$$P_u(k, l) = \sum_{d=d_{free}}^{\infty} a_d \cdot P_d(k, l) \quad (11)$$

where  $d_{free}$  is the free distance of the convolutional code,  $a_d$  is the total number of error events of weight  $d$  and  $P_d(k, l)$  is the probability that an incorrect path at distance  $d$  from the correct path is chosen by the Viterbi decoder as defined by:

$$P_d(k, l) = \begin{cases} \sum_{i=(d+1)/2}^d \binom{d}{i} \rho^i (1-\rho)^{d-i} & \text{if } d \text{ is odd} \\ \frac{1}{2} \binom{d}{d/2} \sum_{i=d/2+1}^d \binom{d}{i} \rho^i (1-\rho)^{d-i} & \text{otherwise} \end{cases} \quad (12)$$

where  $\rho(k, l)$  is equal to  $BER(k, l)$ .

The  $P_e(k, l)$  function is finally used to evaluate  $P_{err}(k)$  with the last equation:

$$P_{err}(k) = 1 - \prod_l (1 - P_e(k, l)) \quad (13)$$

## 6.2 The MAC model

The 802.11 Distributed Coordination Function is used to calculate when to grant access to the transmission medium. While implementing the DCF would have been particularly easy if we had used a recurring timer that expired every slot, we chose to use the method described in [17] where the backoff timer duration is lazily calculated whenever needed, since it is claimed to have much better performance than the simpler recurring timer solution.

The higher-level MAC functions are implemented in a set of other C++ classes and deal with:

- packet fragmentation and defragmentation,
- use of the RTS/CTS protocol,
- rate control algorithm,
- connection and disconnection to and from an Access Point,
- the MAC transmission queue,
- beacon generation,
- etc.

Table 2: Performance of Simulation Core

|                                    |         | <i>GTNetS</i> | <i>yans</i> |
|------------------------------------|---------|---------------|-------------|
| event HOLD (s)                     | avg     | $1.2e^{-5}$   | $1.3e^{-5}$ |
|                                    | std dev | $9.4e^{-8}$   | $9.9e^{-8}$ |
| packet transmission<br>(packets/s) | avg     | 260447        | 423945      |
|                                    | std dev | 15109         | 4151        |
| packet creation<br>(packets/s)     | avg     | 526441        | 1352757     |
|                                    | std dev | 6388          | 13671       |

## 7 Performance And Scalability Considerations

Although our design objectives were to put correctness, API cleanliness, and ease of use on top of the requirements, performance and scalability are also of major concern to us: we want to be able to quickly perform large and complex simulations.

The cpu and memory usage of the core utilities provided by the simulator has been closely monitored and profiled and extensively optimized to allow the simulator to deal with very large numbers of events and packets:

- the memory used by the simulation packets depends linearly on the size of the simulated packets since the simulation packets serialize each simulated header and payload in a correspondingly-sized byte buffer;
- the allocation of packets is done by a PacketFactory which uses a free-list to avoid de-allocating and allocating packet structures constantly; and
- packet buffers are almost always created with the right reserved size because the Buffer class used by the Packet class calculates on the fly the total number of bytes needed by all the protocol headers and trailers during the start of the simulation.

We designed a few micro-benchmarks to evaluate the performance of the resulting APIs: each run of a simulator was repeated 10 times and the average and standard deviation calculated. The results are summarized in Table 2; namely

- the behavior of the event scheduler was profiled on a synthetic workload (a uniform distribution of 320000 elements for the HOLD model) with the stdC++ map scheduler;
- the Packet API was submitted to a simple packet transmission benchmark repeated 1000000 times: a packet is created, payload, UDP, and IPv4 headers are added, the packet is copied once, and the IPv4, UDP and payload are removed; and
- the Packet API was also submitted to a packet creation benchmark also repeated 1000000 times: a packet is created and payload, UDP, and IPv4 headers are added.

The performance of the *GTNetS* and *yans* event schedulers are, rather unsurprisingly, very close since they are both based on the stdC++ map data structure. The Packet data structures, on the other hand, have very different performance characteristics: *yans* can

generate about 60% more packets than *GTNetS* when using an optimized shared library. This performance gain comes mostly from the very small number of memory allocations performed by *yans*: protocol headers can be allocated on the stack and do not require a costly call to the heap allocator.

Of course, one could wonder how accurately these benchmarks reflect real-world use. To answer this question, we performed a 802.11 scenario based on our 802.11 MAC/PHY model. Node A moves away from Node B and saturates the transmission medium with constant-sized packets generated at periodic intervals at the UDP layer. Every simulation second and for 42 simulation seconds, A moves 5 meters away from node B and generates 100000 packets of 2000 bytes each.

The code was built with gcc 4.1.0, full optimizations enabled, asserts disabled, and with static linking enabled. Building *yans* as a shared library on an x86 system generates code which is vastly slower due to the way Position Independent Code is implemented on this platform: our sample simulation scenario is 38% faster when using a static library than when using a shared library. Our IPv4 and UDP stacks do not calculate the IPv4 and UDP checksums by default, which generates slightly incorrect `pcap` output but which saves up to 20% of runtime. The wall-clock runtime of this simulation on an x86 Centrino-based system is just under 15s which means that this simulation creates, and deals with around  $\frac{42 \times 100000}{15} = 280000$  packets/s: this is a bit more than half the theoretical throughput reported by our packet creation benchmark.

## 8 Conclusion

Dissatisfaction with the software design provided by *ns-2* and the inadequacy of the licensing terms of the other existing tools led us to design Yet Another Network Simulator. The new features provided by this simulator have already proven useful to us: our 802.11 models have seen major simplifications and cleanups since we started porting them to *yans* and we hope to be able to add emulation, parallelization, and real-world code integration capabilities easily to this framework.

## 9 Acknowledgments

Hossein Manshaei contributed to the design of the 802.11 PHY model used in *yans* and Thierry Turetletti provided comments on early versions of this paper. M.L. architected *yans*, wrote most of the software, and designed and conducted all of the experiments herein, while T.H. provided *yans* feedback and assisted in writing this paper.

## References

- [1] S. Bajaj, L. Breslau, D. Estrin, K. Fall, S. Floyd, P. Haldar, M. Handley, A. Helmy, J. Heidemann, P. Huang, S. Kumar, S. McCanne, R. Rejaie, P. Sharma, S. Shenker,

- K. Varadhan, H. Yu, Y. Xu, and D. Zappala, "Virtual internetwork testbed: Status and research agenda," July 1998, uSC Computer Science Dept, Technical Report 98-678.
- [2] "Global Mobile Information Systems Simulation Library," <http://pcl.cs.ucla.edu/projects/glomosim/>.
- [3] "NCTUns Network Simulator and Emulator," <http://nsl.csie.nctu.edu.tw/nctuns.html>.
- [4] G. F. Riley, "The georgia tech network simulator," in *MoMeTools '03: Proceedings of the ACM SIGCOMM workshop on Models, methods and tools for reproducible network research*. New York, NY, USA: ACM Press, 2003, pp. 5–12.
- [5] FDK. Fdk usage agreement. [Online]. Available: <http://www-static.cc.gatech.edu/computing/pads/fdk/fdk-usage-agreement.html>
- [6] A. Varga, "The OMNeT++ distrete event simulation system," Software on-line: <http://whale.hit.bme.hu/omnetpp/>, 1999.
- [7] J. Cowie, A. Ogielski, and D. Nicol, "The SSFNet network simulator," Software on-line: <http://www.ssfnet.org/homePage.html>, 2002, renesys Corporation.
- [8] "Java in Simulation Time (JiST)," <http://jist.ece.cornell.edu>.
- [9] "OPNET Technologies, Inc." <http://www.opnet.com>.
- [10] "Scalable Network Technologies, Inc." <http://www.scalable-networks.com>.
- [11] Simplified wrapper and interface generator. [Online]. Available: <http://www.swig.org/>
- [12] P. L'Ecuyer, "Good parameter sets for combined multiple recursive random number generators," *Math. Oper. Res.*, no. 1, pp. 159–164, 1998.
- [13] FSF. Gnu scientific library. [Online]. Available: <http://www.gnu.org/software/gsl/>
- [14] "Executable and linkable format," *TIS standard Portable Formats Specification*, 1993.
- [15] G. Holland, N. Vaidya, and P. Bahl, "A rate-adaptive mac protocol for multi-hop wireless networks," in *Proceedings of the 7th annual international conference on Mobile computing and networking*, 2001, pp. 236–251.
- [16] M. B. Pursley and D. J. Taipale, "Error probabilities for spread-spectrum packet radio with convolutional codes and Viterbi decoding," in *MILCOM '85 - Military Communications Conference*, 1985, pp. 438–441.
- [17] Z. Ji, J. Zhou, M. Takai, and R. Bagrodia, "Scalable simulation of large-scale wireless networks wirh bounded inaccuracies," in *Proceedings of the Seventh ACM Symposium on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, October 2004.



---

Unité de recherche INRIA Sophia Antipolis  
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes  
4, rue Jacques Monod - 91893 ORSAY Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique  
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)

Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

---

Éditeur  
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)

<http://www.inria.fr>

ISSN 0249-6399