Yet Another Network Simulator

Mathieu Lacage
INRIA, Planete Project
2004 route des Lucioles, 06902 Sophia Antipolis, FR
mathieu.lacage@sophia.inria.fr

Thomas R. Henderson
Department of Electrical Engineering
University of Washington, Seattle, WA 98195, USA
tomhend@u.washington.edu

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*, `http://yans.inria.fr/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*.

Categories and Subject Descriptors

I.6.7 [**Simulation Support Systems**]: Environments; C.2.1 [**Network Architecture and Design**]: Wireless communication

1. INTRODUCTION

This paper reports on the design and goals of a new discrete-event network simulator for Internet research. The name of the simulator (Yet Another Network Simulator, or *yans*) and of the title 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 [2]. 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 appropriately for our project (compatible with the GPLv2).

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 `tclcl` 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 another part as OTcl methods. This problem is amplified when inheritance is brought into the equation since both the OTcl and the C++ methods of the parents are inherited. Experience has shown that debugging in this environment can be a challenge.

Table 1: Simulator licensing terms

GloMoSim/QualNet	Academic until 2000 and commercial since then.
GTNetS	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.

Besides *ns-2*, a number of other open-source simulators have been developed, including GloMoSim [8], NCTUns [14], *GTNetS* [17] (including the RTIKit library [7]), OMNET++ [22], SSFNet [5], and JiST [12]. Two popular commercial tools are OPNET [15] and QualNet [18].

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. GOALS AND REQUIREMENTS

2.1 Licensing

We intend 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 contributes 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 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[1] 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 [19] have been designed to make this easier.

The single-language core of *yans* 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 does 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 into *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.

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 and 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

[1]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.

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 a negligible performance penalty on Python simulations compared to pure C++ simulations.

4.1 Memory management

Because the C++ language does not enforce any specific memory management policy, it is easy to get lost in the set of possible solutions and to build a system where various components use different incompatible policies which leads to the dreaded question: *Shall I delete that pointer or not?*

To avoid these problems, we decided to enforce the following rules whose design goal is to allow us to make decisions on memory management only with knowledge of the local properties of the code under scrutiny:

- If you can, pass around object values rather than pointers to objects or references to objects. C++ object values do not need to be explicitly managed by the programmer: they are automatically copied and destroyed as needed by the compiler and the runtime. The potential cost of manipulating object values rather than object pointers or object references comes from the higher cost of copying an object value. However, the fact is that we usually access the fields of an object much more often than we pass and copy it around which means that the cost of copying the object a few times is often much smaller than the cost of accessing the fields of the object many times. The point here is that managing the ownership of object values is much easier than managing the ownership of object references or pointers and the truth is that the cost of accessing the fields of an object is the same in both cases.
- Try to avoid objects which require an implementation of `operator =`, of a copy constructor, and of a destructor.
- Do not pass around object references unless explicitly required for the implementation of a copy constructor or an `operator =` method. This makes it easier to understand the `in` vs `out` vs `inout` semantics of an argument to a function since it eliminates the ambiguity of reference arguments which have pointer semantics and which look like value arguments to client code. Otherwise, a user reading the client code would have to locate the function signature which, usually, is located far away from the actual use, to determine whether or not reference semantics are active.
- Invoke delete only on the pointers on which you have invoked new yourself or which were returned from a factory function (whose name should contain the *create* keyword).

While these rules allow us to solve the problem we set out to solve, that is, the need for a uniform, not too inefficient, and not too complicated system, they also introduce problems of their own. The main problem they introduce is the need for the user himself to perform a large number of deletes since, when the simulation is completed, he needs to destroy every object created during the scenario topology construction.

In most cases, he could decide to ignore these deletes since his program is about to terminate anyway and the memory will be reclaimed automatically by the operating system. However, it is clear that some users will need to perform these deletes, if only to ensure that they have no other memory leaks when their program terminates. A simple solution to this problem would be to require our users to use a smart pointer similar to the `boost::shared_ptr` (see [3]) smart pointer and stop using raw pointers. The smart pointer would then be responsible for destroying the objects it points to when needed.

So far, we have resisted the temptation to add a rule requiring the use of Boost[3] or TR1[21] because we wanted to make it as easy as possible to build our software on a large set of platforms with a minimum set of dependencies. However, comments from some early users shows that the use of raw pointers does not scale very well: future versions of *yans* will thus include extra memory management rules forbidding the use of raw pointers as much as possible and encouraging the use of a boost-like `shared_ptr` smart pointer.

4.2 The event scheduler

The event Scheduler provides a classic event scheduling API that allows its users to look up 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. 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-entrance problems.
- 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. In Boost/TR1 lingua, events are also known as fully-bound functors.

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 accross a large range of hardware and software platforms and sometimes leads to very hard to de-

bug 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 easy to plug in the simulator new scheduling algorithms (such as [4] and [20]), the complexity performance of the insert and the remove operations are not specified. The currently implemented algorithms provide:

- Linked List: O(n) insert and O(1) remove;
- Binary Heap: O(log(n)) insert and remove; and
- stdC++ map: O(log(n)) insert and remove.

4.3 Functors

While the Functor design pattern is already used by the Event class, it is also used by the Callback class, an implementation of the non-bound Functor pattern, very similar in spirit to the Boost/TR1 functor template and inspired by the *Generalized Functors* chapter from [1].

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.

4.4 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;
- 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 multi-rate PHY model based on piecewise SNIR calculations with an energy threshold; and
- a periodic traffic generation model which can be connected to both UDP and TCP sockets.

IPv4 routing is currently limited to static routing since we did not implement any IPv4 routing daemon responsible for updating the content of the routing table. It is our hope that such features will be provided by the port of an existing routing daemon to the simulator through the use of the techniques described in section 7.

5. PACKETS

The way the simulator represents simulated network packets has a major influence on:

- *performance and scalability*: the number of simulation packets we can create, process, and destroy per real second,
- *emulation*: conversions between simulation and simulated packets must be performed at the simulator/real-world boundary,
- *real-world code integration*: conversions must also be performed to and from the format expected by the real-world code, and
- *ease of use*: how easy it is to implement a model which needs to add and remove a new type of protocol header to a packet.

To understand the design of the *yans* packet data structure and the necessary compromise we had to make, we outline the architecture of the packet data structure used in *ns-2* and *GTNetS* and detail their impact on the above-mentioned features.

5.1 ns-2 Packets

The Packet data structure in *ns-2* is an aggregate of protocol headers: each protocol header must be registered with the tcl runtime through the tcl function `add-packet-header` which adds the size (in bytes) of the new protocol header to the maximum number of bytes which can be used by a packet.

Whenever a user wishes to create a Packet, he needs to invoke `Packet::alloc` which returns a pointer to a new Packet instance. This pointer can then be passed around and the memory associated with it is reclaimed only when the user calls `Packet::free`.

`Packet::alloc` allocates a byte buffer of exactly the maximum size calculated by `add-packet-header` and reserves consecutive bytes for each type of protocol header registered at that time. The data of each protocol header is stored at a constant offset (calculated at runtime) from the start of the byte buffer of the packet. The C++ users who wish to access the individual fields of a protocol header packed in this packet need to use a `HDR_PROT` macro (where `PROT` designates the name of the protocol) which returns a properly-casted pointer into the packet's byte buffer at the right offset.

The *Common* protocol header is a mandatory header which is always present in *ns-2* packets. It contains information on:

- the type of the last protocol header added to the packet,
- the total size of the packet,
- a uid for the packet,
- on-the-side information for various routing protocols, and
- miscellaneous random stuff added over the years.

Printing the content of a Packet is left to external tracing classes which are responsible for printing the content of the last protocol header added to the packet (as indicated by

the common header's type).

Adding a new protocol header is thus a matter of allocating a protocol header type by adding an entry to the `packet_t` enum, registering the protocol header with the tcl function `add-packet-header`, adding a C++ macro definition `HDR_PROT`, and, adding printing code to the tracing classes to handle the newly allocated header type. This process is probably one of the biggest problems with this design: it is necessary to edit the main `packet.h` header to add a new type of protocol header and edit the tracing code to deal with it correctly which means that outside developers must develop patches against the ns-2 core rather than simply write code which hooks itself into the simulator core.

This design has a number of other interesting shortcomings and we outline some of them below:

- It is not possible to encapsulate a protocol into itself: (such as IP over IP): only one instance of a protocol header can be stored into a packet instance;
- It is only possible to know the type of the last protocol header added to a packet: the various protocol stacks are responsible for updating the type of the *common* header as they *peel* the packet, that is, remove headers;
- There is no provision to create, represent, and re-assemble Packet fragments;
- Packet ownership must be carefully managed to avoid memory leaks, or, worse, double frees.

5.2 GTNetS Packets

GTNetS Packets export a much higher level interface than *ns-2*: each Protocol Header is represented as a subclass of the `PDU` base class and must provide serialization and deserialization methods. These methods are used during parallel simulations to convert simulation packets to and from simulation messages between computing nodes.

`PDU`s can be pushed and popped in and out of Packets through the `Packet::PushPDU` (add to top of stack) and the `Packet::PopPDU` (remove from top of stack) methods. When a PDU is pushed, the Packet makes a deep copy of that PDU through its `PDU::Copy` method. When a PDU is popped, the packet returns a pointer to the PDU stored internally. This means that its users do not need to free the PDUs returned by a call to `Packet::PopPDU` but also that the PDU returned stays valid only until the next call to `Packet::PushPDU` (which frees the last PDU popped to make room for the newly-pushed PDU) or until the packet is itself deleted.

The content of each Packet can be easily printed for debugging or tracing purposes with the `Packet::DBPrint` method which only has to loop through the PDU stack to invoke the `Pdu::Trace` method on each PDU.

Packets are allocated with C++ `new` and destroyed when not needed anymore by C++ `delete`. They are passed around as pointers.

This design obviously solves the biggest problems identified in the previous section for *ns-2* packets: it is very easy to add a new type of `PDU` into a packet by creating a subclass of the `PDU` base class. The Packet itself does not require any modification to handle that new type of PDU. It can also easily perform double encapsulations (IP over IP) but there is still no provision for fragmentation support or ownerhip management of Packets.

5.3 Yans Packets

In *yans*, a Packet can be instantiated with a call to the `Packet::create` method which returns a raw Packet pointer. This raw pointer is expected to be stored in a `PacketPtr` or a `PacketConstPtr` object instance (although we do not force our users to do so) which are both reference-counting smart-pointers: they automatically destroy the packet only when there are no users left. These smart pointers behave like raw pointers which is very important for typical user code: they can be compared against zero and it is possible to assign zero to them. Indeed, typical packet queues return zero when no packets are left in the queue and typical MAC-level retransmission algorithms set the `current` pointer to point to the packet being currently transmitted and set it to zero when the packet is successfully transmitted or dropped.

Protocol headers are represented by subclasses of the base class `Chunk` (not named Header because it can also represent a protocol trailer). Each such subclass must implement three methods: `Chunk::add_to`, `Chunk::peek_from` and `Chunk::remove_from`. `add_to` is expected to reserve enough space in the byte buffer and serialize its data into the reserved space, `peek_from` is expected to deserialize its data from the byte buffer, and `remove_from` is expected to trim the reserved space from the byte buffer. The byte buffer is represented by an instance of the `Buffer` class which is inspired by the BSD/Linux mbuf/skb data structures: it is possible to efficiently reserve or trim an arbitrary number of bytes from the start or the end of the byte buffer.

The serialized representation of the header could be anything but we expect it to be the *exact* network representation of the header. If this is done, then, the in-memory representation of a Packet is a buffer of bytes which contains a real network packet: simulation packets are nothing but nice shiny wrappers around network packets.

yans Packets also provide a way to attach arbitrary on-the-side information to each packet through subclasses of the abstract base class `Tag`: we hope that this mechanism will allow us to avoid uncontrolled random additions to the main Packet class (which happened to the *ns-2 common* header but also to the *GTNetS* Packet class to support NIX routing for example).

This design makes it completely trivial to support packet fragmentation, reassembly, and repacketization since a simulation fragment maps naturally to a network fragment: the `Packet::copy (start, end)` method can create fragments and they can be concatenated through a call to either the `Packet::add_at_start` or the `Packet::add_at_end` methods.

The solution outlined above provides a nice and elegant solution to the problem of generating and dealing with packet fragments, and also to some of the problems of emulation and real-world code integration: conversion to and from real-world packets is very easy since the simulation packets already contain a bit-for-bit accurate representation of the network packets. Generating *pcap* traces out of such simulation packets is also trivial since it is a matter of prepending the correct *pcap* headers to each simulation packet before dumping them into a file.

Generating this bit-for-bit representation as opposed to doing a simple copy of the data through a memcpy was evaluated to be about 20% slower (10% for the serialization and 10% for the deserialization) on the performance benchmarks described in section 9. Another issue to consider with this design is that a packet has no knowledge of which protocol headers were added to it: it is up to the user to decide how

to interpret the bytes stored in the packet. This makes it impossible to print automatically in a human-readable form a packet's content: the user must peel and print each header separately.

Despite the limitations outlined above, we believe that the resulting data structure is both reasonably efficient (it is, rather unexpectedly, faster than the *GTNetS* data structure as detailed in section 9), easy to use for our users, and makes emulation and real-world code integration more natural than alternative designs such as *GTNetS* or *ns-2* where a Packet contains a discrete list of protocol headers.

6. TRACING

One of the distinctive features of Network Simulation tools is that they allow us to trace exactly what is going on in the simulated network: it is easy to instrument every part of the protocol stack but it is much harder to extract meaning out of the vast amount of information generated during a single simulation.

The tracing framework of *yans* was designed with the goal of being able to selectively enable various trace event sources, potentially in a large or small number of network nodes, and gather them together easily.

Each model is responsible for registering each of its trace event source into a `TraceContainer` and for generating the trace events themselves. For now, *yans* defines four types of trace event sources:

- *packet logging*: whenever a packet drop, or a packet transmission is logged, 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.
- *stream logging*: whenever a model writes to a stream with `operator` <<, the write is forwarded to another user-provided stream.
- *function call logging*: whenever a function or method is called, it is forwarded to a user-provided callback with the same signature.

Each trace source stored in a TraceContainer can be individually connected to any user-provided trace event consumer. For example, the 802.11 PHY layer traces each PHY-level state change and any user could log into a text file each of these state changes.

The ability to individually select each event source is very important to be able to minimize the amount of data logged into a file during simulation but it is also important to be able to analyze this potentially large data set. To do this, *yans* provides a small python-based post-processor application that can display on screen a timeline of the events logged and allows simple zooming in and out of the timeline. It is also possible to dump to a png image file the currently displayed portion of the log.

In the future, we plan to integrate this post-processing visualization tool in a more ambitious C++ based user interface that would provide tools to layout network topologies, configure the traffic sources and the trace points, run the simulation, and post-process the simulation traces.

7. REAL-WORLD CODE INTEGRATION

Our interest in being able to integrate pre-existing 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 (most notably user-space routing daemons), and we believe that the ability to simulate real-world networks accurately with their real-world bugs and limitations is key to facilitate the impact of contributions from the research community on real-world protocols. Practitioners will also benefit from an easy-to-use tool to conduct protocol development, debugging, and tuning.

Integrating pre-existing real-world code poses a number of very interesting technical challenges which we have spent considerable time attempting 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.

To obtain multiple versions of the same set of global variables in the same process, we plan to use the properties of Position Independent Code on ELF[6] systems. On these systems, PIC code must access its global variables as constant offsets relative to the address of the code itself: thus, mapping the code twice at different base addresses will make each copy of the code use a different set of global variables. Mapping multiple times the same code at different base addresses can be done by using the `dlmopen` function exported by the dynamic loader library: this function can load a dynamic shared library in new symbol resolution namespaces with the `LM_ID_NEWLM` flag. All we have to do to use this feature is to recompile our applications as PIC code and link them into shared libraries rather than link them into executable binaries.

Early prototyping work to implement this solution was succesful and allowed us to identify a single shortcoming: the number of new namespaces `dlmopen` can create is limited to 15 on GNU/Linux systems by default because a statically-sized array (of size 16) is used to store namespace data. This means that we would be unable to create more than 15 process contexts on these systems. Increasing this bound is a matter of recompiling the host glibc with a change to the proper `#define` but it would be much more interesting to make the glibc dynamic loader use a dynamically-sized array of namespace data. Work is underway to implement this solution and integrate it in future versions of the GNU libc.

An alternative approach[10], which, among others, solves the problem of static variable context, and was already demonstrated for ns-2 involves automatic kernel source code editing through a C parser so that it may execute in the ns-2 simulation environment. The main drawback of this approach is that automatically parsing and editing source code is inherently much more fragile than reusing the source code as-is.

Providing a full build, link, and runtime environment to user-space code is a matter of providing a new implementa-

tion of a standard C library with POSIX support. We also need to port an existing standard C++ library to this new C standard library to be able to run C++ code within the simulator. The greatest challenge here is the implementation of a proper standard C runtime. For example, global variables such as `errno` are usually allocated in Thread Local Storage (to allow each thread to have a distinct copy of the variable). This means that these variables (marked with the `__thread` keyword) cannot be accessed with the classic ELF PIC techniques: the memory they are stored into must be allocated by the thread library upon the creation of each thread, the data structures used to access them must be initialized by the dynamic loader whenever a thread is created and the runtime must export a `___tls_get_addr` function which is used by the compiler-generated code to access these variables.

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.

8. 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.

8.1 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 [9], 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. This model could be easily improved to take into account the Capture effect (see [13]) and we plan to add a new PHY model where signal attenuation and propagation would be modelized by a simple SNR-based synchronization threshold and where interferences would be modelized by an SNIR threshold.

The PHY layer can be in one of four states:

- TX: the PHY is currently transmitting a signal on behalf of its associated MAC
- SYNC: the PHY is synchronized on a signal and is waiting until it has received its last bit to forward it to the MAC.
- CCA_BUSY: the PHY is not in the TX or SYNC but the energy measured on the medium is higher than Energy Detection Threshold (as defined by the Clear Channel Assessment function mode 1).
- IDLE: the PHY is not in the TX or SYNC states.

When the first bit of a new packet is received while the PHY is in the TX or the SYNC states, the packet received is dropped. Otherwise, if the PHY is IDLE or CCA_BUSY, we calculate the received energy of the first bit of this new signal and compare it against our EDThreshold. If the energy of the packet k is higher, then the PHY moves to SYNC state and schedules an event when the last bit of the packet is expected to be received. Otherwise, the PHY either switches to CCA_BUSY if the total energy received reached EDTreshold or stays in IDLE state. In both cases, the packet is dropped.

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:

$$P_l(d) = P_l(d_0) + n10log_{10}(\frac{d}{d_0}) \quad (1)$$

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), λ 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.

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

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

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

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

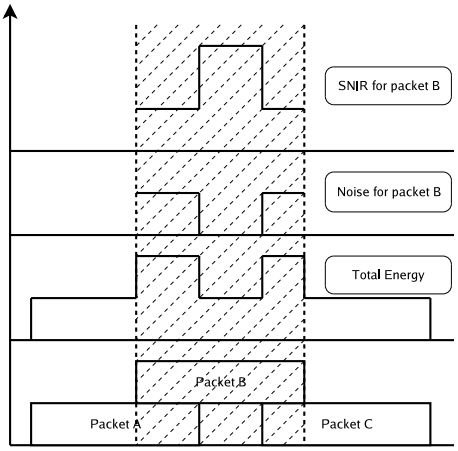

Figure 1: SNIR function over time

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:

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

B_t is the unspread bandwidth of the signal (that is, 20MHz for 802.11a) and $R_b(k,t)$ is the bitrate 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 that 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 Additive White Gaussian Noise 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].

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

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:

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

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:

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

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

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

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

8.2 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 [11] 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.

9. PERFORMANCE AND SCALABILITY CONSIDERATIONS

Although our design objectives were to put correctness, API cleanliness, and ease of use on the 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, 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 `Packet::create` 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.

Since we were not interested in evaluating the performance and accuracy of the models implemented in *yans* but instead in the performance of the core APIs, we chose to design a few micro-benchmarks and to compare our performance to that of *GTNetS* because its architecture is broadly similar to that of *yans* which makes the comparison much more meaningful than comparing *yans* to *ns-2* for example.

Each run of a simulator was repeated 10 times and the average and standard deviation of the execution time of a run 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

Table 2: Performance of Simulation Core

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

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 an 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.

10. 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. *yans*'s moderate use of templates (simplifying the use of the Functor design pattern), coupled with the careful design of its Packet and Tracing APIs can potentially provide a long-lasting architecture and survive many years of abuse by more or less innocent users. *yans* is also unique in that it was designed from the ground up to make it as easy as possible to integrate real-world code and as easy as possible to connect in real-time with real networks: the ability to bridge our simulation tools with the real world systems and implementations of various protocols will improve simulation realism and traceability to similar experiments conducted on emulated and operational networks.

Bringing *yans* from the state of a solid prototype to the state of a widely used mainstream simulator is not possible without more contributions and resources: since most of the stated goals of the *ns-3* project seem to coincide with those outlined for *yans*, it is our belief that some of the components of *yans* can be lifted and integrated in the larger *ns-3* project.

11. ACKNOWLEDGMENTS

Hossein Manshaei, Yongho Seok, and Martin Heusse contributed to the design of the 802.11 PHY model used in *yans*, Steve Mc Canne provided comments on section 7, and Thierry Turletti reviewed early versions of this paper. M.L. architected *yans*, wrote most of the software, and designed and conducted all of the experiments described herein, while T.H. provided *yans* feedback and assisted in writing this paper.

We would like to thank our anonymous reviewers for their detailed comments.

12. REFERENCES

[1] A. Alexandrescu. *Modern C++ Design*. Addison-Wesley, 2001.

[2] 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.

[3] The boost library. http://www.boost.org.

[4] R. Brown. Calendar queues: a fast 0 (1) priority queue implementation for the simulation event set problem. *Communications of the ACM*, 31(10):1220–1227, 1988.

[5] J. Cowie, A. Ogielski, and D. Nicol. The SSFNet network simulator. Software on-line: http://www.ssfnet.org/homePage.html, 2002. Renesys Corporation.

[6] ELF. Executable and linkable format. *TIS standard Portable Formats Specification*, 1993.

[7] Fdk usage agreement. http://www-static.cc.gatech.edu/computing/pads/fdk/fdk-usage-agreement.html.

[8] Global Mobile Information Systems Simulation Library. http://pcl.cs.ucla.edu/projects/glomosim/.

[9] 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*, pages 236–251, 2001.

[10] S. Jansen and A. McGregor. Simulation with real world network stacks. 2005.

[11] Z. Ji, J. Zhou, M. Takai, and R. Bagrodia. Scalable simulation of large-scale wireless networks with bounded inaccuracies. In *Proceedings of the Seventh ACM Symposium on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, October 2004.

[12] Java in Simulation Time (JiST). http://jist.ece.cornell.edu.

[13] A. Kochut, A. Vasan, A. U. Shankar, and A. Agrawala. Sniffing out the correct physical layer capture model in 802.11b. In *ICNP '04: Proceedings of the Network Protocols, 12th IEEE International Conference on (ICNP'04)*, pages 252–261, Washington, DC, USA, 2004. IEEE Computer Society.

[14] NCTUns Network Simulator and Emulator. http://nsl.csie.nctu.edu.tw/nctuns.html.

[15] OPNET Technologies, Inc. http://www.opnet.com.

[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*, pages 438–441, 1985.

[17] 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*, pages 5–12, New York, NY, USA, 2003. ACM Press.

[18] Scalable Network Technologies, Inc. http://www.scalable-networks.com.

[19] Simplified wrapper and interface generator. http://www.swig.org/.

[20] K. L. Tan and L.-J. Thng. Snoopy calendar queue. In *32nd conference on Winter simulation*, pages 487–495, 2000.

[21] Iso/iec tr 19768: C++ library extensions tr1. http://www.open-std.org/jtc1/sc22/wg21.

[22] A. Varga. The OMNeT++ discrete event simulation system. Software on-line: http://whale.hit.bme.hu/omnetpp/, 1999.