UNIVERSITE DE NICE-SOPHIA ANTIPOLIS

ECOLE DOCTORALE STIC
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

THESE

pour obtenir le titre de

Docteur en Sciences

de l'Université de Nice-Sophia Antipolis

Mention: Informatique

presentée et soutenue par

Mathieu LACAGE

Outils d'expérimentation pour la recherche en réseaux

Thèse dirigée par Walid DABBOUS

soutenue le 15 novembre 2010

Jury:

M. Luc Deneire	Professeur	Président
M. Olivier Bonaventure	Professeur	Rapporteur
M. Luiz Felippe Perrone	Professeur	Rapporteur
M. Thomas Henderson	Professeur	Examinateur
M. Jean-Marie Gorce	Professeur	Examinateur
M. Dejan Kostic	Professeur assistant	Examinateur
M. Walid Dabbous	Directeur de Recherche	Examinateur

Acknowledgments

I would like to first thank all the people at the INRIA who made it possible to start and to pursue this PhD thesis to its completion: Gerard Giraudon, Michel Cosnard, Janet Bertot, and David Rey were all kind enough to let me finish writing this thesis quietly.

I am indebted to my adviser, Walid Dabbous and to Thierry Turletti who were both instrumental in starting the project that led to this thesis and who encouraged me until today.

Martin Ferrari and Alina Quereilhac deserve special thank-you notes for their friendship and their implementation of the Nepi concepts.

I am especially grateful to the ns-3 team in general and Sally Floyd and Tom Henderson in particular for allowing me to jump on board of the ns-3 project and for making these past few years so interesting and productive.

There must have been wine involved because I can't remember the details but Jean Michel Dalle is the person who started many years ago the chain of events that led to this manuscript: thank you is not enough but this is all I can offer here.

Finally, thank you to Alexandra, Julie, and Chloe for being there.

Summary

In this thesis, we consider the problem of optimizing the experimentation workflow of a typical network researcher. Specifically, we first identify the need for new experimentation tools which deal with the lack of realism of network simulations in general and which improve the lack of reproducibility, as well as the poor analysis and deployment facilities found in testbeds and field experiments. We then focus on the design and implementation of two new tools to close this gap between simulations on one side and testbed/field experiments on the other side.

First, we integrate within the ns-3 core facilities transparent support for real-time simulation capabilities to allow arbitrary simulation models to be interconnected with the real world and then we build upon this groundwork to create NEPI, an experimentation tool which can automate the deployment of mixed experiments comprising simulations and testbed virtual machines.

Second, we extend the ns-3 network simulator with a new Direct Code Execution module which is capable of executing within the simulator existing user space as well as kernel space protocol implementations and we demonstrate that the resulting system is both extremely robust and one order of magnitude more efficient CPU and memory wise than previous alternatives.

These two new tools radically extend the scope of the network experiments that can be conducted with off-the-shelf hardware by allowing the use of the same protocol implementations in both simulation and testbed environments, hence paving the way to more realistic simulations, more controllable testbeds, and simplifying the switch between one and the other.

Keywords: discrete time event-driven simulation, emulation, real-time simulation, direct code execution, network experimentation

Résumé

Dans cette thèse, nous nous intéressons a l'amélioration du flot de travail d'expérimentation d'un chercheur en réseaux. Pour ce faire, nous mettons en évidence la nécessite d'augmenter le réalisme des simulations réseaux ainsi que de diminuer les difficultés de déploiement et d'analyse des tests sur le terrain ou en environement plus controlé.

Nous avons ainsi commencé par intégrer au sein du coeur du simulateur ns-3 le support transparent pour des simulations temps réel afin d'inter-connecter l'ensemble de ses modèles avec le monde réel. Puis, nous nous sommes appuyés sur cette fonctionalite fondamentale pour développer Network Experimentation Programming Interface (NEPI), un nouvel outil d'expérimentation capable d'automatiser le déploiement d'expériences mixtes comprenant à la fois des éléments simulés en temps réel et des éléments du monde réel. Enfin, nous avons mis au point un nouveau module ns-3 capable d'exécuter directement au sein du simulateur des implémentations existantes de protocoles réseaux venant à la fois d'applications de niveau utilisateur et de la pile réseau du noyau d'un système d'exploitation. Nous avons démontré que notre approche est extrèmement robuste et un ordre de grandeur plus efficace que les solutions existantes du point de vue de l'utilisation Central Processing Unit (CPU) et mémoire.

C'est l'utilisation simultanée de ces deux nouveaux outils qui permet l'utilisation de la même implémentation d'un protocole réseau à la fois en simulation et en tests sur le terrain. Elle représente donc l'étape indispensable vers des simulations réseaux plus réalistes, des tests plus facilement contrôlables et un passage aisé de l'un à l'autre.

Mots clés: simulation évènementielle à temps discret, émulation, simulation temps-réel, expérimentation réseau

Contents

Chapter 1

Introduction

Although users have always found the means to interact remotely with computers since they existed, it was the creation of the ARPANET in 1969 that really started the phenomenal growth of computer networks we still witness today. Since then, many generations of computer and network technologies have been designed, used, and retired but many more are still in use: this explosion in heterogeneity and complexity is what gave way to the rise of the Internet Protocol (IP), the protocol family of choice to enable inter-networking between mobile smart phones, computing clusters, large mainframes, and inter-planetary spatial probes.

Despite IP's success as the major inter-networking layer, the inter-operability between hardware and software devices deployed on existing and future networks is still the major challenge that must be dealt with on a daily basis by every protocol designer, network operator, and network user.

For a long time, network researchers had relatively few tools to help them in this task: the size and the cost of the computer and networking hardware was so high that it made it impractical for anyone but a privileged few to experiment with new protocols before they could be deployed. Simple analytical models and simple simulations that focused on the steady-state equilibrium of a network were the tools of choice to study the impact of a proposed modification on a protocol. This changed drastically by the end of the 1990s and the early 2000 years when the cost of common off the shelf hardware plummeted making it possible to build cheap powerful experimentation platforms such as [7] to investigate both the steady-state of a system and more complex intermittent cross-layer phenomena.

Nowadays, after a decade of hardware cost decreases, it is not only possible to build testbed platforms cheaply but it has become feasible to use small-scale deployments in the field to get even closer to the complexity of the real world. Furthermore, simulation tools have also started to catch up in terms of realism, hence increasing even further the number of options available to researchers for conducting their experiments.

1.1 Experimentation tools

To illustrate the conundrum now facing these researchers, we consider the example of a content distribution protocol running over Transmission Control Protocol (TCP)/IP within a wireless ad hoc network whose task is to ensure that every wireless device in that network gets a copy of a large binary file initially available only in one of these devices. In this case, the objective might be to minimize the total time needed to fully distribute the file or maybe to ensure that no device depletes its battery. The objective could also be a complex combination of these objectives together with a few extra others such as making it possible to perform voice calls over this network while the file is being distributed. Whatever the objective chosen, the problems remain the same:

- the behavior of each protocol layer present in each device is going to influence the behavior of all other layers in ways that are hard to predict precisely. For example, if TCP starts to send data too fast, it will clog the shared wireless spectrum, hence increasing the probability of transmission collisions and thus decreasing the probability of successful packet transmissions which will in turn mean that TCP must retransmit each lost packet and thus further increase contention for access to the shared wireless medium, etc.

- some elements of the system are inherently unpredictable and variable (say, the surrounding wireless interference level) yet they must be estimated since they might dramatically change the behavior of the system being studied.

1.1.1 Field Experiments

Given the number and the complexity of the factors involved in this example, most researchers would immediately turn to a field experiment to study the behavior of the system.

Those with an engineering background would probably start by implementing a simple content distribution protocol and then collect traces about its behavior within a small-scale network made of less than twenty nodes that can be assembled cheaply with off-the-shelf laptops. Then, they might try to observe, analyze, and identify the various cross-layer interactions at play here to revise their content distribution protocol and maybe change slightly the behavior or the parameters of their TCP stack.

Others might instead try to first characterize the wireless background interference level and its impact on packet loss, then move on to model packet transmission interference as a function of the average medium usage level and finally, implement a content distribution protocol that makes use of these models to optimize the target metrics.

While one or even both approaches might be successful at finding a practical solution to the problem at hand, they will also necessarily highlight the considerable hidden cost involved in having to deal with the lack of control and reproducibility of such a field experiment. For example, making sure that the poor performance of a specific version of the content distribution protocol is due solely to that protocol rather than a transient change in the background interference level during that experiment because an undergraduate student used a microwave oven will require a lot of analysis work so that statistically relevant results can be extracted out of the noise.

1.1.2 Testbeds

The complexity of controlling all relevant parameters in a field test is precisely why so many users often turn to setting up testbeds where they can measure or control every parameter of interest and easily reproduce the same experiment over and over again.

In the case we consider here, some researchers will try to build a testbed where wireless interferences are tightly controlled. A Faraday cage would be sufficient to isolate their wireless nodes from the outside world while controlling the interference level between each node in their testbed would require removing the wireless antennas and replacing them with cables fed to a set of signal mixers and splitters. If our experimenters are also willing to perform field experiments to observe typical background interference conditions and generate similar background noise in their testbed, the resulting testbed might become the source of both reproducible and realistic results.

In the end, though, when our researchers turn to the fine interaction between some of the higher layers of the protocol stack and the Medium Access Control (MAC) wireless layer, they might realize how difficult it is to debug and extract meaningful information from protocols implemented in real hardware or within a live Operating System kernel and they might finally see the value of running a simplified experiment within a simulator to be able to observe and trace the state of the entire protocol stack.

1.1.3 Simulations

In the context of this wireless experiment, different layers of the protocol stack will require different simulation techniques. The analog parts of the system (the transmission and reception antennas and the propagation medium of the radio signal between them) could be simulated by solving a set of differential equations using a Finite Element Method [2]. On the other hand, the digital components that make up our experiment, that is, the entire protocol stack from the application to the MAC layer is a system whose state changes discontinuously at discrete time points making it a natural candidate for discrete time event-driven simulations.

Discrete time event-driven simulations

The objective of a discrete time simulation is to compute the values of each state variable of our system within the simulated time frame. Two major techniques are used to do so: time-stepped simulations [3] and event-driven execution runtimes [1]. Time-stepped simulations divide the total simulation time into a series of equal-sized time steps and then incrementally increase the simulation time by this time step value. At each time step, the simulation runtime computes the new values of each state variable. This approach is very simple to implement and can offer great performance when there is at least one state variable that changes its value at each step. However, it is little used because it is hard to come up with a time step value that is sufficiently small to minimize the rounding error of and event's expiration time to the time of the closest previous step and that is big enough to ensure that there is at least one state variable change within each time step.

In practice, discrete simulations are more often implemented by event-driven execution runtimes [1]: rather than compute the new value of each state variable at each time step, the runtime instead computes the new values only when something meaningful happens in the system being modeled. Each interesting event in the real system is modeled by a simulation event to which we associate a timestamp and a state update function. The simulation runtime then proceeds to execute the state update function of each event in the order of increasing timestamps. To illustrate this algorithm, the following pseudo code assumes that the variable `m_events` is a linked-list of events sorted by increasing time stamps:

```
time = 0;
```

```
while (!m_events.empty ())
  {
    Event event = m_events.front ();
    m_events.pop_front ();
    time = event.time;
    event.function ();
  }
```

Network event-driven simulations

While many events can happen in the real world, network simulations usually model only those that are related to the creation and processing of the packets that store the information that must be exchanged between hosts interconnected by a communication network. Typical events include the start or the end of a packet transmission or reception, the expiration of a timeout, or a direct user interaction such as plugging in a network copper cable/optic fiber or starting the download of a file.

A network simulation thus mostly contains models to create and process packets. For example, to model the behavior of the finite-sized transmission buffer located between a TCP stack and a wireless transmitter, we merely need to create events for the arrival and the departure of packets in that buffer. It is then possible to derive through simulation the probability of packet loss due to the buffer being full when a new packet arrives as a function of the arrival and departure patterns.

To be able to easily observe and analyze the detailed interaction between the TCP and the MAC wireless layer of our content distribution system, our researchers would need to create and implement a set of models for each layer, and then instantiate them once for each device of our wireless network. While these models might not accurately reflect the behavior of the underlying system, setting up, executing, and tracing the exact sequence of events that happens during such a simulation is orders of magnitude easier than using a testbed or a field experiment where the protocol implementations run inside a closed hardware device, in an Operating System Kernel, and on distributed physical systems. A certain amount of realism might have been lost but control, reproducibility, and the ability to debug the system have been increased considerably.

Beyond perfect reproducibility and excellent debugging capability, another good reason to use simulations in the example we consider here is cost: while the cost of the hardware devices needed to setup a small-scale field test or testbed is very low, it is still much cheaper to use a similarly-sized simulation on a single machine or a very large simulation on one of the many computing clusters that are readily available to every researcher.

1.1.4 Emulation

Despite their many positive characteristics, simulations are often seen as lacking realism because they are based on the design of models which attempt to define simple abstractions of the real world. A common approach to deal with this issue, yet maintain the excellent reproducibility and debugging capability of typical simulations is to extend these simulations with Direct Code Execution (DCE) capabilities to execute within a simulator the real protocol implementations generally used in testbeds or field experiments.

1.1.5 Mixed experiments

Another way to scale up the size of an experiment cheaply, maintain a high degree of realism and increase controllability and reproducibility is to use a mixed experimentation environment where a testbed, a field experiment, and a simulation synchronized to a real-world clock are used together. The simulation can be used to scale up the size of the experiment in terms of the number of network devices present and provide a high degree of control and reproducibility while its interconnection with a testbed or a field experiment ensures that a certain degree of realism is maintained.

Mixed experiments are, however, rarely used in practice because the complexity of setting them up is very high: users must configure compatibly both their testbed and their simulation which makes it impossible to use the native setup capabilities of each experimentation tool and thus requires a lot of manual work.

1.2 Problem

Many more variants of field experiments, testbeds, and simulators could be profitably used to study the example we have described here but their characteristics would all be fairly similar. Table 1.1 extends upon [4] to present a synthetic view of these characteristics.

Simulation	Emulation	Mixed experiment	Testbed	Field test
− Model based	− Model based	+/− Real system	+ Real system	++ Real system
− Algorithm	+ Real code	+/− Real code	+ Real code	++ Real code
− Simplified	+ Accurate	+/− Accurate	+ Accurate	++ Accurate
+ Insight	+ Insight	+/− Insight	− Black box	−− Black box
++ Scalable	+ Scalable	+/− Scalable	− Not scalable	−− Not scalable
++ Fast	+ Fast	+/− Fast	− Slow	−− Slow
++ Flexible	+ Flexible	+/− Flexible	− Not flexible	−− Not flexible
++ Repeatable	++ Repeatable	+/− Repeatable	− Variable	−− Variable
++ Cheap	++ Cheap	+/− Cheap	− Expensive	−− Expensive
++ No sysadmin	++ No sysadmin	+/− Sysadmin	− Sysadmin	−− Sysadmin
++ No deployment	++ No deployment	−− Deployment	− Deployment	−− Deployment

Table 1.1: Characteristics of existing experimentation tools

The most striking conclusion that can be drawn from this table is that the *perfect* tool does not yet exist which means that users interested in exploring the behavior of a complex system must use more than one tool: a field test will provide difficult to reproduce but realistic data while the corresponding simulation could be used to explore repeatedly the same experiment with much less realism.

The natural solution to this problem would preferably be to use either emulation or mixed experiments: these experimentation environments can be useful on their own thanks to their useful combination of realism, reproducibility, scalability, *etc.* but their most important feature is that they are sufficiently close to full-blown testbeds or field tests to make it easy to switch to them when very high realism is needed. Comparatively, pure simulations suffer from their lack of model realism and the use of software abstractions that are so different from the real world that simulated network protocols must be re-implemented prior to field or testbed experiments.

In practice, though, two important factors contribute to make it hard to use emulation and mixed experimentation environments: mixed experiments are extremely complex to configure, setup, and deploy and emulation is always limited to constrained systems where the size and complexity of the glue layers needed to adapt the simulation environment to the protocol runtime environment is minimal.

1.3 Contributions

In this thesis, we demonstrate the practicality of addressing both issues by attacking the problem from multiple perspectives:

- First, we have implemented the core facilities needed to make ns-3 the first network simulator to support transparently and efficiently the automatic conversion of network packets to and from simulation objects, hence paving the way to transparent support for real-time simulation in every ns-3 model.
- Second, we have designed a flexible unified experiment description model which is the key to allow the automated setup and deployment of mixed experiments which involve a real-time simulation, a testbed, and a field experiment.
- Third, we have radically extended the scope of the emulation tools' capabilities by integrating within ns-3 a Direct Code Execution (DCE) framework which encompasses both user space and kernel space protocol implementations written in C or C++ for Linux. Contrary to previous work, our solution is both highly efficient and extremely robust, able to deal with arbitrary protocol constructs.

While we originally intended to build these new features within the Yet Another Network Simulator (Yans) [5] prototype network simulator, we seized the opportunity of joining the ns-3 team whose objectives were similar to our own to broaden considerably the impact of our work. This is how a large fraction of the software modules we developed during this thesis are now in use by many network researchers around the world.

1.4 Outline

Before considering what sets ns-3 apart from other network simulators, chapter 2 gives an introduction to its design and its architecture and highlights the work that went into making ns-3 a robust, full-fledged network simulator that supports many realistic models and that is used by a growing user community.

In chapter 3, we describe the ns-3 packet facility that is used to store the protocol data exchanged between network entities. We outline how our focus on simulation realism shaped its implementation and how its main distinctive feature, that is, its native transparent support for the conversion back and forth between real network packets and simulation packets is instrumental to enable the Direct Code Execution facility described next and the transparent support for real-time simulations used by NEPI [6].

We build upon this groundwork in chapter 4 to extend considerably the scope of the realism of ns-3 through the integration within the simulator itself of arbitrary linux user space applications as well as the complete Linux kernel network stack. We introduce in this chapter a new lightweight virtualization technology which we coin *User space virtualization* based on the use of compiler-generated ELF PIC assembly code and an ad hoc program loader to supplement the deficiencies of the default GNU C library ELF loader. We then describe and evaluate the CPU and memory performance of the wrapper layers used to provide the emulation of the user space and the kernel space APIs necessary to embed Linux applications together with the linux kernel network stack within the simulator.

Finally, chapter 5 considers the problem of extending the scope of a realistic testbed towards larger scale experiments: we outline how real-time simulations can be used as emulators within an existing testbed to deal with this issue and then illustrate the feasibility of this approach in the case of a simple virtualization testbed based on linux network namespaces interconnected to an ns-3-based emulator. Because the complexity of manually setting up such a mixed experiment is daunting, we finally proceed to describe how we designed and implemented NEPI, a tool used to describe, and automatically deploy and configure mixed experiments.

Bibliography

[1] Edwin Chong. Discrete event systems: Modeling and performance analysis. *Discrete Event Dynamic Systems*, 4:113–116, 1994. 4

[2] R. Courant. Variational methods for the solution of problems of equilibrium and vibrations. *Bulletin of the American Mathematical Society*, 49:1–23, 1943. 4

[3] Yang Guo, Weibo Gong, and Donald F. Towsley. Time-Stepped Hybrid Simulation (TSHS) for Large Scale Networks. In *Proceedings of INFOCOM*, pages 441–450, 2000. 4

[4] Georg Kunz, Olaf Landsiedel, and Georg Wittenburg. From Simulations to Deployments. In *Modeling and Tools for Network Simulation*, pages 83–97. 2010. 6

[5] Mathieu Lacage and Thomas R. Henderson. Yet another network simulator. In *WNS2 '06: Proceedings of the 2006 workshop on ns-2: the IP network simulator*, page 12. ACM, 2006. 7

[6] Mathieu Lacage, Martin Ferrari, Mads Hansen, Thierry Turletti, and Walid Dabbous. Nepi: using independent simulators, emulators, and testbeds for easy experimentation. *SIGOPS Operating Systems Review*, 43:60–65, January 2010. 8

[7] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. *SIGOPS Operating Systems Review*, 36(SI):255–270, 2002. 2

Chapter 2

ns-3: A New Simulation Tool

The objectives pursued in this thesis called for a network simulator that would fulfill many requirements. First and foremost, this network simulator should provide a high degree of realism so that it is easy to either use it as a real-time emulator in a larger testbed or to execute directly within the simulator existing real-world protocol implementations.

Our experience with using other network simulation tools led us to focus on a number of important features that are critical to ensure that this simulator becomes widely used by many researchers for a long time:

- It should favor the creation of a thriving community of network researchers able to contribute to its evolution through the addition of new models and able to take over the maintenance, validation, and verification workload of existing models.
- It should provide a solid software architecture that can withstand extensions and abuse from many different contributors over long periods of time.
- It should make it possible to perform complex simulations. Furthermore, commonly-used scenarios should be easy to describe and run.

Sadly, we realized early on that none of the existing network simulators could fulfill all of these requirements together and that we would have to first invest considerable resources in the development of a new simulation engine before we could focus on the components necessary to achieve our long term goals of a realistic simulation framework.

In this chapter, we provide some historical background on how ns-3 came to be, and we report on its core software architecture. Before discussing some of the technical details of its low-level interfaces, we review first in section 2.1 the other simulators that we considered when we started working on this thesis. We describe in section 2.2 the steps we took to ensure the long-term survival of this project, and then, discuss in 2.3 the salient features of the ns-3 discrete time event-driven simulation engine. Section 2.4 outlines how the ns-3 object model provides a unified interface to common simulation tasks without forcing every model implementation to adopt a rigid structure. Finally, section 2.5 describes the basic architecture of the ns-3 network models and highlights its close relationship to the structure of the network stack found in any UNIX system.

2.1 Not Invented Here

In the 2004/2005 time frame, when we started to work on this thesis, we first considered a few Java-based network simulators such as JiST [9], J-SIM [19] and SSFNet [11] but we eliminated them from consideration on the basis that we wanted to be able to directly simulate existing C/C++ socket-based routing daemons and TCP/IP stacks. We felt that providing a tight level of integration between such C/C++ codebases and a Java simulation core would be too painful. Technologies such as Java Native Interface (JNI) might have made it possible to switch back and forth between the C and the Java runtime but we believed that it would be more productive to focus on a C/C++-only solution to minimize the size of the necessary adaptation layer.

Although commercial simulators such as Qualnet [27] and Opnet [16] were available under favorable terms to academics, and they clearly had sufficient financial resources and backing to be able to live on over the kind of time frame we had in mind, we perceived them as being unwilling to encourage the rise of an open and thriving user community able to meaningfully engage, contribute and influence the development of the tool.

The strongest contender back in 2005 appeared to be Objective Modular Network Testbed ++ (OMNeT++) [28] because it had a reasonably-sized user and contributor community and more favorable licensing terms for commercial users, but the component-based programming model was judged to be incompatible with our code reuse objectives.

ns-2 [3] was the more natural choice because of its considerable existing user community, fully open source nature, and because we had personal experience with this simulator and its previous incarnation ns [25] in research. We also knew about many of its weaknesses; most notably, the software had never been designed with long-term maintenance in mind (lack of coding and documentation standards, lack of maintainable validation and verification tests), the split-object model was difficult to debug, and the Tool Command Language (Tcl) programming language was becoming less popular. The level of abstraction in some of the core ns-2 objects such as packets and addresses hindered our model realism, emulation, and code reuse goals.

For a short while, we considered creating a C++-only simulator from the ns-2 codebase: initial prototypes showed that it was possible and we might have been able to pull off a slow refactoring of the entire codebase in that direction, but this would have required that we rewrite all Tcl models in C++ or drop them altogether and we felt that doing so would negate the whole point of re-using an existing codebase with its models. We also did not feel that we had the resources to maintain an ns-2 backward-compatibility layer.

When it became clear that none of the simulators available would match our requirements, we started the development of the prototype simulator Yans [21]. Yans was never widely used outside of its core developers but the experience we gained while implementing it was critical to validate a number of key software architecture elements.

Eventually, in 2005, when the design and initial development of ns-3 started, we did let go of Yans and started to contribute to ns-3. This decision was not very easy because we were concerned that we would waste too many resources developing the low-level guts of a new simulator without guarantee that it could become useful outside of our small development team. However, in hindsight, there is nothing to regret: the efforts invested in building a solid simulation core paid off considerable productivity, stability and ease of use rewards later when we started to integrate existing real-world protocol implementations in ns-3 through our Direct Code Execution environment (see chapter 4).

2.2 A thriving contributor community

As hinted at above, when the development of ns-3 started, one of the central questions which was discussed at length revolved around how we could make sure that the tool we were building would be useful to others beyond our small development team. Specifically, we did not know how we could encourage the creation and the growth of a community of network researchers willing to contribute to its long-term evolution and maintenance to ensure its survival and continued usage after our initial development funding dried out.

It is hard to claim that we have already been successful in creating a thriving community of contributors who will be able, in time, to take over the maintenance of ns-3 as a whole but there are continuously-improving metrics which show considerable progress towards that goal. For example, figure 2.1 highlights the gradually-increasing total number of users currently registered on the ns-3 user mailing-lists.

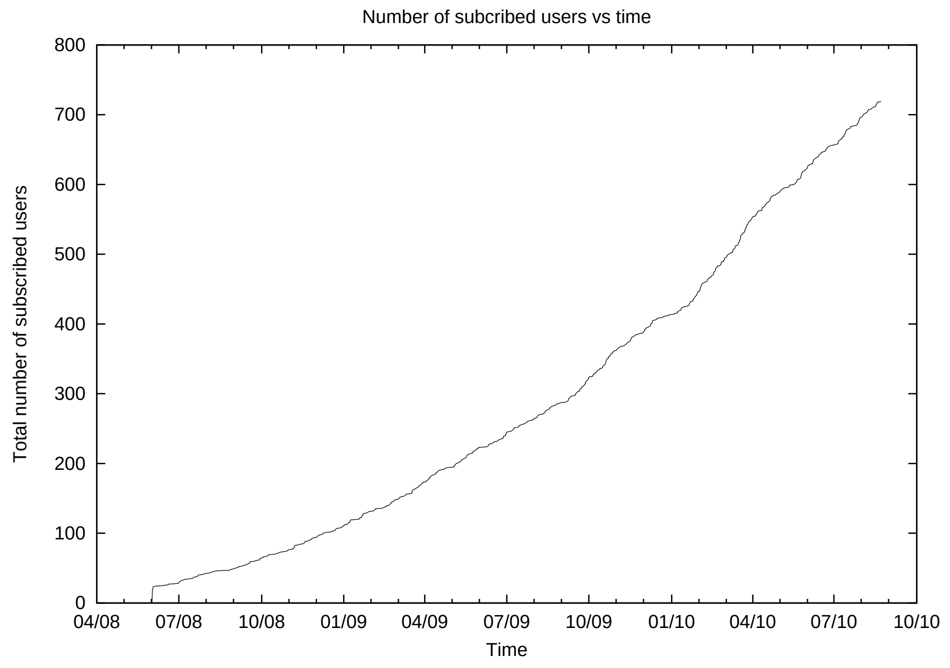


Figure 2.1: Number of users subscribed to the ns-3-users mailing-list as a function of time

Books such as [14] now cover extensively every aspect of the problem of creating a successful open source community but we report here on the most useful and costly steps we took towards this goal in the hope that this serves as a helpful experience for other researchers intending to replicate this approach in other projects.

A time-based release process

As soon as the overall ns-3 architecture settled down, the project moved away from its pre-alpha state to a regular time-based release process with one release every three months: right after every release, a new development cycle starts with the merging of new major de-stabilizing features in the main development source tree. This tree is then slowly frozen until the new features are fully documented, and their tests pass on every hardware and software we support, hopefully in time for the end of the development cycle.

While the above is necessary to allow the ns-3 maintainers to ship stable high-quality releases, the predictability and regularity of this cycle is also critical to allow external contributors and, more generally, every maintainer of a module to plan their own development schedules and synchronize on the merge window. Rather than try to forcibly merge unfinished modules before the freeze deadline, our users know that if they are not ready for the current merge window, they can aim for the next merge window that is only a couple of weeks or months away.

Originally, this process was established solely as a software engineering technique to manage our own internal development tasks but the side-effect of its predictability and regularity is to facilitate and thus encourage contributions to ns-3.

Transparent communication

Another key step we took towards encouraging new contributors to join the project was to forbid the use of private email correspondence or hallway discussions for technical or management matters and to always use our public development mailing-lists. We thus occasionally exposed publicly a few flame-wars and strongly-worded technical discussions but the temporary embarrassment due to these was more than offset by the useful input that many users contributed to these discussions.

While the amount of time spent conducting these discussions over low-bandwidth email rather than face to face is daunting, we feel that being so open early on is what convinced many of our early serious contributors that it was worth joining the project.

2.3 Discrete time event-driven simulations

As pointed out in section 1.1.3 where we described the fundamental concept behind discrete time event-driven simulations, it is easy to implement a toy simulation runtime that can process events. However, the implementation found in a real simulator is considerably more complex due to the need to export this functionality behind a programming interface that is easy to use and that does not significantly affect performance.

We already discussed in section 1.1.3 the event execution main loop but there are other operations that are fundamental to a discrete time event-driven simulator:

- **Run**: execute every event in order of increasing timestamps until there are no more events or the user triggers **Stop**.
- **Stop**: set the global stop flag to true to make sure that the **Run** function returns at the next available opportunity.
- **Now**: return the current simulation time.

- **Schedule**: create a new event, insert it in the global event list, and return a reference to the newly-created event.
- **Cancel**: disable the execution of an event that is present in the global event list. Cancel is usually considered a constant-time operation that merely sets a disabled flag in the event that is checked by **Run** before executing that event.
- **Remove**: Remove an event from the global event list to ensure that it is never executed. This operation is usually considered to be of $O(n)$ complexity (where n is the number of events).
- **Status**: return whether an existing event has expired or is still *running*

In practice, there is very little variance across different simulators in the way these functions are exported to users except on the way the simulation time is represented and on how simulation events are created and managed.

In the next two sections, we focus on these two aspects of the ns-3 simulation runtime library. We first consider the representation of simulation time in section 2.3.1 and then discuss how simulation events are created and managed later in section 2.3.2.

2.3.1 Time

Despite its apparent simplicity, the management of the timestamp associated with each simulation event is far from being trivial as discrete-time simulators must fulfill a number of important requirements. First, a simulator must ensure that the behavior of the same simulation is always exactly the same, regardless of the hardware and software platform it is running on. Specifically, the simulator should ensure that events are never re-ordered and that their expiration time is always the same when moving from one platform to another.

Another important requirement is the need to provide a large range of high-resolution timestamps: some network simulations need nanosecond resolution over periods of time as long as a couple of months, and sometimes a couple of years.

Finally, the performance and ease of use of the Application Programming Interface (API) used to manipulate the user-visible simulation time is also critical since simulation models need to manipulate this simulation time heavily, if only to schedule events and calculate the delay until they expire.

Related work

The most straightforward way to represent simulation time is to use a double-precision floating point type [22]. The 52-bit wide mantissa provided by these 64-bit floats makes it possible to represent $\frac{2^{52}}{1000000\times60\times60\times24\times365} = 142$ years of simulation time with microsecond precision or a bit less than two months when nanosecond precision is needed. Many simulators such as ns-2 [3], Georgia Tech Network Simulator (GTNetS) [15], and early versions of OMNeT++ [28] chose to do so.

While this range is sufficient in most cases, the use of floating point arithmetic introduces many portability problems that hamper our reproducibility objectives: the result of these arithmetic operations depend heavily on the specifics of the underlying CPU floating point unit, Operating System (OS), compiler and runtime library implementation. The excess precision provided by the Intel x86 64-bit floating point representation of doubles is probably the most well-known example of this problem: users who attempt to run their code on an IEEE-conformant 64-bit floating point unit often realize too late that their arithmetic computations implicitly rely on the excess precision provided by x86 hardware and then give up on trying to obtain the same results on different platforms.

Because the semantics of integer arithmetic are much simpler to define precisely, their implementations do not suffer from the reproducibility problems that plague floating point arithmetic. Simulators such as Global Mobile Information Systems Simulation Library (GloMoSim) [29] and later versions of OMNeT++ [28] that are more concerned with better reproducibility over a wide range of hardware and software platforms and that intend to support a larger timestamp range with at least nanosecond resolution thus usually chose instead a 64-bit wide fixed-point integer to represent simulation time. This increases the simulation time range up to 584 years for nanosecond resolution but most critically makes it much easier to obtain the same simulation results on different platforms.

The choices made by the simulators discussed here are summarized in table 2.1.

	64-bit float	64-bit integer	resolution
ns2	✓		second
GTNetS	✓		second
OMNeT++ 3.x	✓		varying
OMNeT++ 4.x		✓	varying
GloMoSim	✓	✓	nanosecond

Table 2.1: Time type in existing simulators

Integral types still need floats

While fixed-point integers provide the foundation on which long-running, reproducible, and efficient simulations can be built, they still suffer from one sizable problem: it is not easy for users to perform even simple computations with integer time values that avoid overflow and still preserve accuracy. For example, to calculate the bandwidth from the number of bytes received and the delay between the first and the last byte received, one might write something along the lines of:

```
int nbytes = ...;
uint64_t delay_ns = ...;
uint64_t bandwidth_bytes_per_s = nbytes * 1000000000 / delay_ns;
```

The above would however give invalid results most of the time since the type of the variable nbytes is `int` (that is, 32bit in general), and multiplying it by one billion is almost surely going to overflow, hence, giving a final result that is meaningless. This code could, of course, be rewritten to trade-off the likely overflow by a less likely underflow:

```
int nbytes = ...;
uint64_t delay_ns = ...;
uint64_t bandwidth_bytes_per_s = nbytes / (delay_ns / 1000000000);
```

which, however, would trigger a division by zero if the delay is smaller than one second.

In general, even simple computations that manipulate time variables are likely to need temporary variables of much higher range or precision than what native integer types can provide, hence leading to either incorrect final results or to computations that are rewritten to use floating point variables as follows:

```
int nbytes = ...;
double delay_ns = ...;
double bandwidth_bytes_per_s = nbytes / (delay_ns / 1000000000);
```

Of course, using floating point arithmetic to perform these computations leads back to the reproducibility problems discussed in previous section.

Another problem commonly observed when using floating point numbers to represent simulation time is that the definition of equality, and more generally, of ordering relationships between timestamps becomes much more complex to state and use correctly. For example, to maximize the reproducibility of the arithmetic operations involving timestamps over a wide range of platforms, two floating point timestamps t_1 and t_2 are often defined as being equal iff $|t_1 - t_2| \leq \delta$ where δ is a small floating point number which depends on the currently-chosen time resolution. In practice, when definitions such as these are used, most model developers ignore them which results at best in unexpected simulation results due to differing interpretations of what being equal means.

Implementation

Because one of the objectives pursued by ns-3 was to ensure that the same simulation running on two different platforms would give the same results, we felt compelled to go beyond what existing network simulators did towards that goal and to provide a means for model developers to ensure that they can perform easily complex computations while avoiding the dreaded problems coming from floating point arithmetic.

The solution which ns-3 settled upon is based on the same 64-bit integral type used by GloMoSim and OMNeT++ to represent simulation time. This simulation time is exported to users through a special class type named `Time`. This class is used to hide the time resolution currently in use: it can be converted to and from seconds, nanoseconds, etc. Similarly to the OMNeT++ `SimTime` class, `Time` overloads the C++ addition, subtraction, and comparison operators to improve the readability of typical user code. Typical use-cases are shown below:

```
Time t = Seconds (10.2);
Time ns = NanoSeconds (1003);
if ((t - ns) >= MicroSeconds (10))
```

```
  {
    // do something
  }
int64_t us = t.GetMicroSeconds ();
```

However, instead of following the choice made by OMNeT++ of overloading the multiplication and division operators and implementing them with floating point arithmetic that lose precision and range, ns-3 provides an implicit conversion operator from `Time` to the new integral type `int64x64_t`.

`int64x64_t` is a fixed point integral type that supports a 64-bit integer part and a 64-bit fractional part (the `x` letter used in its type name denotes the position of the fractional point). It overloads all the C++ arithmetic operators (+, -, *, /, ⟨, etc.) and implements them by performing only integer computations. On 32bit systems, it uses a software implementation of 128-bit arithmetic operations that was originally written for early versions of the Cairo library [2]. On 64-bit systems, it attempts instead to use the much faster gcc extensions for 128-bit arithmetic that are based on the `__int128_t` type.

This new facility trivially provides the same computation results on every platform, while making it possible for users to perform arithmetic operations whose final result depend critically on intermediate variables with fractional elements. For example, to calculate an approximation of 0.5×500000 (500kbit/s during 0.5s gives 250kbits transmitted), a user willing to risk getting different results on different platforms due to floating point arithmetic could write:

```
double a = 0.5;
double b = 500000;
double c = a * b;
```

A user who cares about getting the same result on every platform might try to hardcode himself a constant multiplication factor with am integer type too small for his data (say, 1000 with 32-bit integers):

```
int32_t a = 500;
int32_t b = 500000000;
int32_t c = a * b / 1000; // oops: overflow
```

This user might be smart-enough to use a big-enough integer type (say, 64-bit):

```
int64_t a = 500;
int64_t b = 500000000;
int64_t c = a * b / 1000; // right result !
```

But, with ns-3 he can trivially write the simpler code shown below to obtain the same correct result on every platform:

```
int64x64_t a = 0.5;
int64x64_t b = 500000;
int64x64_t c = a * b;
```

Avoiding conversions to floating point variables is now easy which makes it possible to rewrite the bandwidth calculation discussed in previous section as follows:

```
int nbytes = ...;
Time delay = ...;
int64x64_t bandwidth = nbytes / delay.GetS ();
```

The complexity of the resulting 64.64 arithmetic operations are well hidden from the user but their CPU efficiency on 32-bit CPUs is potentially problematic due to the use of a software-only 128-bit arithmetic implementation. Until hardware devices which sport hardware optimized 128-bit arithmetic units (devices such as every modern 64-bit CPU) are more widely adopted, it is important to quantify the overhead introduced by the use of this facility which is the topic of the following section.

Performance evaluation

To evaluate the overhead introduced by our use of 128-bit integer arithmetic we first consider a micro benchmark designed to capture the CPU cost of `int64x64_t` arithmetic operations relative to that of simpler `int64_t` and `double` operations. This benchmark estimates the average runtime needed to execute one arithmetic operation by measuring the runtime of a loop that repeats the same operation one billion times. The source code was compiled with the highest optimization level (`-O3`) and the generated assembly was checked to ensure that the inner loop does contain only the instructions that correspond to the operation of interest. This measurement was repeated at least ten times until the relative standard deviation drops below 0.05. Table 2.2 reports the result of this benchmark for the four primitive arithmetic operations applied on each of the types of interest on both a 32bit and a 64-bit system.

		add (ns)	cmp (ns)	mul (ns)	div (ns)
32-bit CPU	`int64_t`	1.08	1.83	6.69	12.0
	`double`	1.36	1.48	2.25	2.63
	`int64x64_t`	5.37	4.90	88.1	119
64-bit CPU	`int64_t`	0.67	1.46	2.17	17.1
	`double`	1.31	1.48	2.17	2.60
	`int64x64_t`	3.13	1.32	13.3	88.0

Table 2.2: Average runtime per 64.64 arithmetic operation on a 32bit and a 64-bit processor

While table 2.2 highlights the considerably higher cost on both 32bit and 64-bit systems of performing a 64.64 arithmetic operation instead of the equivalent `double` or `int64_t` operations, in practice, there is no direct relationship between the overhead measured here and the overhead observed in a real simulation because these computations represent only a fraction of the total amount of work done during any simulation.

Equation 2.1 illustrates how we can estimate from the per-operation costs reported in table 2.2 the total simulation runtime that was spent performing these computations:

$$Tarith = N_{add} \times t_{add} + N_{cmp} \times t_{cmp} + N_{mul} \times t_{mul} + N_{div} \times t_{div} \tag{2.1}$$

For every example simulation scenario present in ns-3, we thus measured its total runtime ($Ttotal$) and the number of operations performed. We then estimated the time spent performing these arithmetic computations with `int64x64_t` ($Tarith_{64.64}$) and compared it with the time that would be spent if we used instead `double` arithmetic ($Tarith_{double}$). Table 2.3 presents these results for the four simulation scenarios that exhibited the highest estimated relative overhead defined by equation 2.2

$$Toverhead_{est} = 100 \times \frac{Tarith_{64.64} - Tarith_{double}}{Ttotal} \tag{2.2}$$

Hardware	Scenario	$Toverhead_{est}$ (%)	$Ttotal$ (s)
32bit	`wimax/wimax-multicast`	5.27	0.57
	`wimax/wimax-ipv4`	4.27	0.29
	`wireless/mixed-wireless`	3.24	0.16
	`csma/csma-star`	2.01	2.23
64-bit	`wimax/wimax-multicast`	1.08	0.36
	`wimax/wimax-ipv4`	0.84	0.19
	`wireless/mixed-wireless`	0.56	0.12
	`csma/csma-star`	0.48	1.49

Table 2.3: Relative estimated overhead for scenarios that exhibit the highest estimated overhead

On 64-bit systems, when the `__int128_t` facility is available, the overhead of the extra reproducibility provided by `int64x64_t` is less than one percent, and thus negligible. While the software implementation used on 32bit systems is less efficient and can incur up to five percent overhead on the macro-level benchmarks conducted here, this overhead is still within what is acceptable given the higher safety and reliability obtained.

2.3.2 Event management

While being able to describe and manipulate the simulation time is important to be able to execute a discrete time event-driven simulation, it is also necessary to be able to create events, associate with them a piece of code to execute when they expire, and return a reference to them that can later be given back to the **Cancel**, **Remove** and **Status** functions.

A C implementation

In a C-based simulator, the only way to represent an arbitrary piece of code would be to use a function pointer. The example below shows how this might be done:

```
struct Event {
  void (*function) (struct Event *);
  Time timestamp;
  bool canceled;
};
void Schedule (Time delay, struct Event *event);
```

To use the above, a user would have to create his own event type:

```
struct MyEvent {
  struct Event ev;
  // extra fields here.
};
static void my_function (struct Event *ev) {
  struct MyEvent *event = (struct MyEvent *)ev;
  // do my thing here
}
struct MyEvent *event = malloc (sizeof (struct MyEvent));
event->function = &my_function;
Schedule (Seconds (10), event);
```

A C++ implementation

In C++, the above is somewhat easier to do and can be rewritten as:

```
class Event {
public:
  virtual void Expire (void) = 0;
private:
  Time m_timestamp;
  bool m_canceled;
};
void Schedule (Time timestamp, Event *event);
void Cancel (Event *event);
void Remove (Event *event);
```

so that users can derive from the Event base class and provide their own version of the Expire method:

```
class MyEvent : public Event {
  virtual void Expire (void) {
    // do my thing here
  }
};
MyEvent *event = new MyEvent ();
Schedule (Seconds (10.0), event);
```

Stop and wait ARQ

A common problem with the approach considered so far is that it forces the user to create one subclass for each kind of event that can happen. To illustrate how painful this can be, we consider the case of a simple stop-and-wait Automatic Repeat Request (ARQ) [26] protocol that implements error control by waiting for an Acknowledgment (ACK) packet after sending each data frame: if the ACK timeout expires before an ACK is received, the sender retransmits the data frame and waits again for the ACK. This protocol requires the sender to deal with three types of events: **DataReceived**, **AckReceived** and **AckTimeout**.

Whenever we receive data, we need to send back an ACK:

```
class DataReceived : public Event {
  StopAndWaitArqProtocol *m_src;
  StopAndWaitArqProtocol *m_dst;
  Data *m_data;
  virtual void Expire (void) {
    AckReceived * ack = new AckReceived ();
    ack->m_src = m_dst;
    ack->m_dst = m_src;
    ack->m_data = m_data;
    Time delay = ...;
    Schedule (delay, ack);
  }
};
```

Whenever we receive an ACK, we need to cancel our ACK timeout and send the next packet present in our transmission queue:

```
class AckReceived : public Event {
  StopAndWaitArqProtocol *m_src;
  StopAndWaitArqProtocol *m_dst;
  Data *m_data;
  virtual void Expire (void) {
    if (m_dst->m_currentData != m_data) {
      // we arrived too late ?
    } else {
      Cancel (m_dst->ackTimeout);
      Data *next = m_dst->m_txQueue->GetNext ()
      m_dst->Send (next);
    }
  }
};
```

If we are not lucky and the ACK is not received before the ACK timeout expires, we need to retransmit our data:

```
class AckTimeout : public Event {
  StopAndWaitArqProtocol *m_sender;
  virtual void Expire (void) {
    m_sender->Send (m_protocol->m_sender);
  }
};
```

Finally, to send data, we first start the ACK timeout and then create a receive event for the receiver.

```
class StopAndWaitArqProtocol {
  friend class AckTimeout;
  friend class AckReceived;
  friend class DataReceived;
  StopAndWaitArqProtocol *m_other;
  void Send (Data *data) {
    m_currentData = data;
    AckTimeout *timeout = new AckTimeout ();
    timeout->m_sender = this;
    Schedule (Seconds (1), timeout);
    Time delay = ...;
    DataReceived *received = new DataReceived ();
    received->m_src = this;
    received->m_dst = m_other;
    Schedule (delay, received);
  }
};
```

The problem with this approach is that we are forced to split the entire protocol implementation in many subclasses which need to manipulate both state that is local to themselves and that is shared with the protocol instances communicating together.

ns-2, GTNetS, OMNeT++

To alleviate this pain, existing simulators such as ns-2, GTNetS and OMNeT++ [3, 15, 28] split the Event class in two:

```
class Event {
  Time m_timestamp;
  bool m_canceled;
};
class Handler {
  virtual void Invoke (Event *event) = 0;
};
Schedule (Time delay, Handler *handler, Event *event);
```

which makes it possible to rewrite the stop and wait ARQ example as follows:

```
class StopAndWaitEvent : public Event {
  enum Type type;
  StopAndWaitArqProtocol *src;
  StopAndWaitArqProtocol *dst;
  Data *data;
};
class StopAndWaitArqProtocol : public Handler {
  virtual void Invoke (Event *ev) {
    StopAndWaitEvent *event = (StopAndWaitEvent *)ev;
    switch (event->type) {
    case ACK_TIMEOUT:
      // ...
      break;
    case ACK_RECEIVED:
      // ...
      break;
    case DATA_RECEIVED:
      // ...
      break;
    }
  }
};
```

This programming interface reduces the amount of boilerplate necessary to implement typical protocols and centralizes the entire protocol event handling behind a single entry point which generally improves its maintainability and readability. However, it does so at the cost of storing all the information about different events in a shared data structure (the `StopAndWaitArqEvent` class in this example). In simple cases like here, when most events use the same fields with the same semantics, this works fine but for more complex protocols, when the number of events grows and the number of fields that are useful to only a subset of these events grows, figuring out which fields should be used by each event becomes a real problem.

Forwarding event subclasses

Another option that no existing simulator uses but that nicely deals with the problem outlined above involves the use of forwarding `Event` subclasses. Instead of splitting the `Event` base class in an `Event` plus subclassable `Handler` class, we can instead implement one subclass per event type that forwards the event to the protocol implementation. For example:

```
class AckReceived : public Event {
  AckReceived (StopAndWaitArqProtocol *ackReceiver, Data *data);
  virtual void Expire (void) {
    m_ackReceiver->AckReceived (data);
  }
};
class StopAndWaitArqProtocol {
  void DataReceived (Data *data) {
    ...
    AckReceived *ack = new AckReceived (m_other, data);
    Schedule (delay, ack);
  }
  void AckReceived (Data *data) {
    ...
  }
};
```

The need to implement one forwarding `Event` subclass per event type increases the amount of boilerplate code that users need to write but it makes it possible to clearly separate the data specific to each event while still allowing the entire protocol implementation to be centralized in a single class, hence maximizing the readability and maintainability of complex protocol implementations.

The ns-3 implementation

To improve upon the forwarder `Event` subclass solution described above, ns-3 uses C++ templates to generate automatically their code. In the case of our ARQ example, the amount of boilerplate code becomes zero. `Schedule` now takes a variable number of arguments: the first argument is still the delay until the event expires, the second argument is a pointer to the member method to invoke when the event expires, the third argument is the object on which the member method should be invoked and all subsequent arguments are forwarded as-is to the target member method.

```
class StopAndWaitArqProtocol {
  void DataReceived (Data *data) {
    ...
    m_ackTimeout.Cancel ();
    Schedule (delay, &StopAndWaitArqProtocol::AckReceived,
              m_other, data);
  }
  void Send (Data *data) {
    m_ackTimeout = Schedule (m_timeoutDelay,
                             &StopAndWaitArqProtocol::AckTimeout,
                             this, data);
    Time delay = ...;
    Schedule (delay, &StopAndWaitArqProtocol::DataReceived,
              m_other, data);
  }
  void AckReceived (Data *data) {
    ...
  }
};
```

In simple protocol implementations, it is of course still possible to keep the switch-based code organization that GTNetS, OMNeT++, and ns-2 enforce but when the protocol complexity increases, it is easy to split the event handlers in separate functions with different arguments. The following illustrates the case where a user would want to emulate the facility provided by these less flexible simulators:

```
class StopAndWaitArqProtocol {
  void Handle (enum StopAndWaitArqEventType type, Data *data) {
    switch (type) {
    case DATA_RECEIVED:
      m_ackTimeout.Cancel ();
      Schedule (delay, &StopAndWaitArqProtocol::Handle,
                m_other, ACK_RECEIVED, data);
      break;
    case ACK_TIMEOUT:
      ...
      break;
    case ACK_RECEIVED:
      ...
      break;
    }
  }
};
```

While the complexity of the implementation of these template-based forwarding events (based on the ideas developed in [7] about generalized bound functors) might scare away more than one C++ programmer, their flexibility and ease of use make them ideal to implement very simple protocols as well as complex systems that use many kinds of events.

2.4 The ns-3 Object model

So far, we have not discussed the more challenging aspects of conducting real simulations that integrate many models together and that try to extract information about the behavior of the underlying protocols: creating and managing events and simulation time is useful but not sufficient. In this section, we discuss some of the other challenges that a real simulation engine needs to tackle to become more useful. We highlight especially the need for these facilities to be accessible in a uniform way, to ensure that the resulting system is easy to use.

2.4.1 Requirements

A network simulation which studies only one specific protocol in isolation such as the stop and wait ARQ protocol described in previous section is very rare. In practice, it is much more common to focus on the complex interactions between two or more protocol layers and thus to integrate together in a single simulation a set of models potentially developed by many independent researchers. The most challenging problem here is that we need to make it *easy* to use these models together: they need to obey a number of common rules to simplify typical simulation tasks such as parameter configuration, trace collection, but also to do so in a way that is still consistent with software modularity and robustness.

Memory management might be overlooked but doing so would ignore the fact that C++ is not a garbage-collected language and that it thus requires the manual management of the memory allocated to each object. Automating this task and making sure that every model used in ns-3 follows the same policy is critical to make it possible for mere mortals to setup and run a complex simulation that does not leak memory.

Modularity is generally seen as a software engineering problem best left to component model freaks. It is however critical to clearly define where the system should be modular and more importantly where it should not to avoid creating a simulator that provides so many abstraction layers that no one really knows anymore how and where to extend it.

Parameter configuration is one of the most common feature used in a simulation engine: when users intend to study the behavior of one or more protocols under varying parameter values, they want to be able to easily control precisely a large set of parameters over a large range of values. For example, it should be easy to specify that the TCP retransmit timeout of node Y should be 200ms and to run many simulations with different values for that parameter to explore its impact on the behavior of the simulation.

Trace collection: of course, it would be somewhat useless to run a simulation without collecting some information about the behavior of the protocols being studied. In some cases, we want to simply dump in a trace file the set of packets that reach a specific point of the protocol stack of a node. In other cases, we need to perform more complex online processing to avoid generating gigabytes or terabytes of trace data on disk.

We explore on these requirements in the remainder of this section.

2.4.2 Memory management

For anything but the simplest simulation, allocating objects on the stack is impossible as they are allocated in factory functions[1] which return references to newly-allocated objects. Using the heap exclusively is thus the only viable solution but it does not come cheap: as pointed out by Yans [21] and GTNetS [15], using raw new/delete pairs in a large complex system gives rise to many questions related to the ownership of the resulting pointers. The central issue is that of figuring out which piece of code is allowed to delete a pointer when more than object contains a copy of that pointer.

For a short while, we considered the use of the Boehm [1] garbage collector as the exclusive means of allocating and releasing memory throughout ns-3 but we decided otherwise for fear of making it harder to integrate other C/C++ libraries in ns-3. In the end, we settled on the use of reference counting: whenever we copy a pointer, we increment a counter

[1]A Factory is a common software Design Pattern [12]: it refers to an object or a function which is responsible for creating instances of a certain type of object, function, or structure.

associated to the object it points to and whenever we destroy a pointer, we decrease the counter. When the counter reaches zero, that is, when there are no pointers pointing to it, the object is deleted. The counter is incremented and decremented through the pair of functions `Ref` and `Unref`. The major advantage of using reference counting is that it is a fairly widespread memory management technique that many developers are already familiar with.

To avoid having to perform a local inspection of each function that copies or destroys a pointer to ensure that it posts the matching `Ref` and `Unref` calls and thus ensure the global correctness of the program, we encapsulate every ns-3 object pointer in a C++ smart pointer which captures every copy and destruction of the pointer to maintain a correct reference count. The sample ns-3 smart pointer template class `Ptr<T>` is based upon the strategy outlined below:

```
class Ptr {
public:
  Ptr &operator = (const Ptr &other) {
    if (m_object != 0) {
      m_object->Unref ();
    }
    m_object = other->m_object;
    m_object->Ref ();
  }
  MyObject *operator -> () {
    return m_object;
  }
private:
  MyObject *m_object;
};
```

To deal with the unavoidable reference cycles that come with a complex simulation experiment, every ns-3 object needs to perform its cleanup in its `DoDispose` method rather than its destructor. To illustrate how this mechanism works, we consider here the classic example of two objects that hold a pointer to each other: if A holds a pointer to B and B holds a pointer to A, their reference count will never reach zero and they will never be deleted. To forcibly break this cycle, we rely on the user to issue at least one call to `Dispose` so that if A is disposed, its pointer to B is destroyed, the reference count of B drops to zero, B is deleted, its pointer to A is destroyed, the reference count of A drops to zero, A is deleted.

In practice, users never need to call the `Dispose` methods directly as they are called automatically by the simulation runtime at the end of the simulation. They thus merely need to implement cleanup in `DoDispose`, chain up to their parent class `DoDispose` method, and make sure they derive from the `ns3::Object` base class to get all of these features.

2.4.3 Modularity

It is mainly our experience with ns-2 that convinced us that we needed to be really careful to ensure that ns-3 would stay modular in the long run and that it would not become overloaded by the numerous models integrated into it over time. Our biggest concern early on was that of dealing with the `Node` class typically used to represent one of the components of a network that contains a CPU, some memory, persistent storage, and a few network interfaces that are connected to other nodes through simulated network links.

In ns-2, the `Node` class accumulated over the years a lot of cruft as every new protocol integrated in ns-2 unconditionally added to this class its node-related data: with time, every model in ns-2 started to use these various fields and it became quickly impossible to remove any of them or figure out which protocol updated which fields.

To avoid reproducing this mistake, we could have chosen to use a component model such as COM [10] or XPCOM [6] to clearly split each piece of functionality in separate interfaces that are navigated with `QueryInterface` using 128-bit Universal Unique IDentifiers (UUID). However, we felt that such a solution was somewhat overkill to deal with the weak base class problem that plagued ns-2 and that it might hamper the ease of use of ns-3 considerably. We thus chose to instead extend the ns-3 `Object` class with a COM-like aggregation facility: figure 2.2 illustrates how three unrelated subclasses of this base class can be dynamically aggregated together by a user with the `AggregateObject` method to form a circular linked list of objects. Once aggregated together, a user can then navigate from one to the other with the `GetObject` method which provides functionality equivalent to the COM `QueryInterface` method.

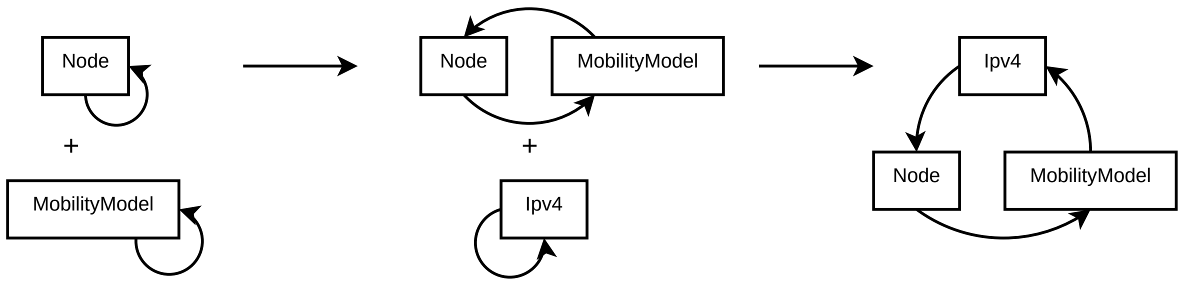


Figure 2.2: Three objects are incrementally aggregated together during the experiment setup

This simple mechanism makes it possible to dynamically attach together a `Node`, and a `MobilityModel` to assign a spatial 3D position to that node without the node itself ever knowing anything about the concept of a position. It is only the other models that are located within the node and that need to access the 3D position that will be aware of it. While this mechanism is very limited and does not provide the scope and breadth of features found in real component systems such as COM or even in the OMNeT++ component-based framework, it has so far proven sufficient to handle all of our extensibility needs hence allowing us to avoid the dreaded ns-2 weak base class problem.

2.4.4 Parameter configuration

As pointed out in section 2.4.1, being able to configure easily the set of parameters provided by each model is another very common task that simulation tools need to handle. While it might be possible to let each model adopt its own approach, this would lead to considerable fragmentation and would make it very hard to keep track of which parameters have been set to which values for anything but the most trivial simulations.

In ns-2, the simulation core makes it easy to initialize C++ class member variables from a value specified in the user Tcl simulation script but this simple mechanism does not provide much control over the value of a parameter in individual objects. To do so, users need to perform the *set default/create/reset default* dance or to obtain a reference to the object once it has been created and explicitly modify its value.

GTNetS, Yans, and GloMoSim provide no support to facilitate this task but most other simulators deal with this issue more extensively. The OMNeT++ component model, for example, makes it possible to control both the default value of an attribute as well as its value in each individual object. Although it is based on a different model which is more hierarchical, the Scalable Simulation Framework (SSF) Domain Modeling Language (DML) provides similar functionality.

In line with OMNeT++ and SSF DML, ns-3 provides extensive control over the attributes of each object. On top of memory management and software modularity, the `Object` base class is also responsible for centralizing metadata about the set of attributes that can be introspected and modified at runtime for each kind of `Object` subclass. This metadata database is used by the following ns-3 tools to solve common simulation tasks:

- `CommandLine`: the ns-3 command-line parsing class recognizes automatically a set of command-line arguments that can be used to change the default value of every attribute in every object type. For example, to set the `ChecksumEnabled` attribute in the `ns3::Ipv4L3Protocol` object to true, we can use the `--ns3::Ipv4L3Protocol::ChecksumEnabled=true` switch and thus obtain IP headers that contain a correct IP checksum rather than the default value of zero.

- `ConfigStore`: this class can be used to introspect the value of all attributes in every object and dump it in an XML or a raw text file. These files can be used as a reference data-set for every simulation run or they can be modified and re-read by `ConfigStore` to re-run a simulation with a precisely set of attribute values.

- `GtkConfigStore` provides a graphical front end to the same facilities and makes it easy to discover the set of attributes available and inspect and modify their values before running a simulation.

- The ns-3 API reference documentation is partly automatically generated to include a list of the attributes of each object together with their default value and a brief description of their purpose.

These features are also available directly to simulation scenario writers when they want to control various attributes directly from their own simulation scripts:

```
Config::SetDefault ("ns3::DropTailQueue::MaxPackets",
                     StringValue ("80"));
Config::Set ("/NodeList/5/DeviceList/1/TxQueue/MaxPackets",
             StringValue ("80"));
```

From the perspective of a model developer, integrating parameters in this global metadata facility is a matter of subclassing the `Object` base class and of defining a `GetTypeId` method that registers a set of attributes for each parameter and returns a `TypeId` instance. For example, to export the member variable `m_ackTimeout` used to control the duration of the timer that is started whenever data is sent in the `StopAndWaitArqProtocol` class discussed in previous section, we could do something similar to this:

```
class StopAndWaitArqProtocol {
public:
  static TypeId GetTypeId (void);
private:
  Time m_ackTimeout;
};
TypeId StopAndWaitArqProtocol::GetTypeId (void) {
  static TypeId tid = TypeId ("StopAndWaitArqProtocol")
    .SetParent<Object> ()
    .AddAttribute ("AckTimeout", "Duration of the ACK timeout",
                   StringValue ("1ms"),
                   MakeTimeAccessor (&StopAndWaitArqProtocol::m_ackTimeout),
                   MakeTimeChecker ())
    ;
  return tid;
}
```

2.4.5 Trace collection

Although configuring the set of parameters used by a set of models is necessary to be able to run any simulation, we have so far ignored the more important problem of how we can extract information from a running simulation either for online processing or offline post-processing.

While offline post-processing is the most flexible and convenient way to extract information from a long-running simulation when we do not know before hand how we are going to analyze its output, it is usually impractical to keep track of every event that happens in the simulator because this would generate trace files that are impossibly large and thus hard to store and hard to post-process.

In certain cases, the sheer amount of information stored in a trace file can make the analysis sufficiently complex that it is easier to filter out the data before it is written to disk or to perform a complete online analysis during the simulation itself. In other cases, online processing cannot be avoided: for example, to ensure that we do not stop a simulation until it runs long enough to reach its steady state.

In both cases, though, the simulation runtime needs to make it easy to collect the information we want and to either process it directly or send it in a specific format to persistent storage. In practice, though, the most common approach to this problem, *print statements*, makes online analysis and trace file formatting impossible.

Print statements

When compilation and link times are sufficiently small, many users succumb the temptation to edit the source code of the model implementations they are using: `printf` and/or `std::cout` are simple low-tech ways to get data from the simulator to persistent storage.

While sprinkling the source code with calls to these functions is very easy to do, it suffers from a number of downsides, some of which are obvious. First and foremost, it rules out any form of online analysis and makes it very hard to change the format of the trace file generated since every call site would need to be changed to adapt to a new format. Second, the time saved when doing this for the first time is usually lost not once, but often many times when a new version of the simulation models is released and the simulation scripts and the `printf` patches must be ported to the new version of the models which have undergone heavy changes independently.

Modifying existing models to add the needed tracing print statements has many other downsides: it makes it harder for a simulator user to remember what has been modified and why. Similarly, unless he uses rigorously a source code version control tool, the simulation scripts become harder to publish since they must be bundled together with the associated trace patches, etc.

Java-based simulators can deal with these issues with the help of Aspect Oriented Programming (AOP) techniques [20] based on the bytecode introspection and modification capabilities of a Java Virtual Machine (JVM) to dynamically insert calls to user-provided functions in arbitrary locations of the models: simulation scripts are standalone and contain descriptions of the trace hooks together with the functions invoked whenever these trace hooks are reached. There is, however, no such solution for C and C++ systems which provide no assembly code introspection capability and thus require ad hoc workarounds.

ns-2, GTNetS

Neither ns-2 nor GTNetS really attempt to provide a solution for online processing and leave the users who care about this on their own.

ns-2 merely tries to (rather unsuccessfully) encapsulate the formatting component of the trace file generation in a separate class so that multiple trace file formatters can be connected to the trace event generators. The set of trace events as well as their arguments are hardcoded and cannot be extended without considerable work.

GTNetS, on the other hand, does not attempt to abstract the trace file format but it makes it easier to filter events by node id, protocol number, and packet content to minimize the complexity and the size of the resulting trace files.

OMNeT++

OMNeT++ stands out from its competitors by providing facilities to support both reasonably complex online and offline processing.

When its models use the OMNeT++ trace event reporting facility, it is possible for users to easily specify one of a set of predefined actions as a handler for any trace event. For example, users can specify that the mean of a specific variable be calculated during the simulation and output at the end. While the set of predefined actions that can be attached to each trace event covers quite a lot of use cases, it is not possible to extend the system through new trace processing actions.

OMNeT++ also provides support for very fine-grained filtering of the data that goes in its trace files: each individual trace event generator can be individually turned on or off.

ns-3

The fundamental tracing abstraction introduced by ns-3 is the distinction between trace sources (models which generate trace events) and trace sinks (functions in user scripts which need to be notified of trace events). Each model is responsible for registering in the ns-3 `Object` metadata database the set of trace sources it exports. The following example is a lstlisting copy of the metadata registration function of the ns-3 TCP implementation: it defines a trace source named *CongestionWindow* which represents the underlying variable named `m_cWnd`.

```
TypeId
TcpSocketImpl::GetTypeId () {
  static TypeId tid = TypeId("ns3::TcpSocketImpl")
    .SetParent<TcpSocket> ()
    .AddTraceSource ("CongestionWindow",
                     "The TCP connection's congestion window",
                     MakeTraceSourceAccessor (&TcpSocketImpl::m_cWnd))
    ;
  return tid;
}
```

The declaration of the `m_cWnd` variable is also straightforward:

```
class TcpSocketImpl {
...
  TracedValue<uint32_t>          m_cWnd;
...
};
```

At runtime, the `Object` base class provides sufficient introspection capabilities to detect which trace sources each object instance supports and allows the connection of arbitrary trace sink functions to the *CongestionWindow* trace source. The trace sink is then called whenever the value of the TCP congestion window changes. It can then perform arbitrary processing before the simulation completes. The following code sample highlights how this facility could be used to generate a simple trace file that contains one line for each value set in the congestion window variable:

```
void MySink (uint32_t oldValue, uint32_t newValue) {
  std::cout << Simulator::Now () << " " << newValue << std::endl;
}
Config::Connect... (
   "/NodeList/0/$ns3::TcpL4Protocol/SocketList/0/CongestionWindow",
    MakeCallback (&MySink));
```

When users do not need access to these low-level trace operations, they can easily enable the default packet-based trace file generators which collect the set of packets being sent or received on an interface in a binary pcap file [5] or a text file.

While the ns-3 trace facility lacks some of the built-in online processing capabilities that OMNeT++ already provides, its flexible tracing framework makes it possible to offload all of these complex online processing responsibilities entirely to users.

2.4.6 Conclusion

Originally, ns-3 did not intend to define or use a very formal object model because we felt that it was important to maximize flexibility for model developers so that they do not have to redesign or rewrite entirely existing models. However, feedback from early users as well as our own experience with implementing or porting simulation models to ns-3 convinced us that we needed nonetheless to provide common facilities that would be easy to use, that would make it possible to easily integrate and use together models that were developed separately, and that would require little to no changes to the core logic of existing models. Over time, the `Object` base class thus slowly and incrementally evolved into a full-fledged framework that focuses on recording in the ns-3 metadata database the functionality provided by each object, hence, making it easy to wrap and export existing functionality in a way that allows users to configure, and trace every model through the same programming interface.

However, the *release early, release often* development philosophy that we adopted means that there is still ongoing work to extend the `Object` class and its associated metadata database:

- The tracing mechanism needs to be extended to provide built-in support for typical online processing algorithms (calculating the mean, variance, etc.) and we intend to extend the metadata database to record for each numerical trace source extra information that describes the default processing that should be applied to the trace source.

- The attribute configuration system needs to be extended to support more transparently arrays and structures.

- The metadata database needs to be extended to record information about which objects can be *connected* together and which methods need to be invoked to connect them so that it becomes possible to both serialize and de-serialize to/from persistent storage the complete description of an experiment: external simulation tools that provide very high-level abstract descriptions of an experiment could then generate one of these experiment description file rather than generate C++ or Python code.

- A generic Graphical User Interface could be built on top of the connection metadata database described above to make it possible to interactively construct a simulation scenario by droping object boxes on a canvas and connecting them together.

2.5 Network models

In section 2.3.2, we touched briefly upon how a typical network protocol could be implemented in a network-oriented event-driven simulator. However, the models that are present in ns-3 cover many more layers of the protocol stack: the most current version of ns-3 sports a complete IPv4, Address Resolution Protocol (ARP), User Datagram Protocol (UDP), and TCP stack. It integrates real-world stacks such as the Linux and BSD network stacks through Network Simulation Craddle (NSC) [18]. ns-3 contains a few simple models to generate and collect traffic but more importantly provides many MAC and Physical (PHY) layers. The oldest such model is a detailed IEEE 802.11 implementation ported over from Yans [21] that features both infrastructure and ad hoc mode, multiple rate control algorithms, interference and propagation loss models, many classic mobility models such as random walk, random direction, etc. More recent additions include a Wimax [13, 17] model, a model of underwater communication systems, and a spectrum-aware channel and PHY modeling framework [8]. The routing layer includes a nix-routing [24] module ported from GTNetS, OLSR [4] ported from ns-2, and AODV [23].

While there would be little point in discussing here the details of how each of these models work and how they can be used, we focus instead in this section on a high-level outline of the ns-3 core network models to illustrate how ns-3 can be extended and where new models can be easily plugged in.

2.5.1 ns-2

During the nineties, when ns-2 was first released, its developers were mostly interested in studying the behavior of TCP over internet-style wired links that can be approximated with relatively simple queues and delay/throughput/jitter models. Consequently, ns-2 defines only high-level abstractions of the network: a `Node` contains solely a 32bit `nodeid` that is used as source and destination address of every packet. Nodes are interconnected by a chain of `Connector` objects, each of which performs some processing on its incoming packets before forwarding them to the incoming port of its outgoing connector. For example, to model a simple wired link that interconnects two applications on two nodes, one might assemble the topology shown in figure 2.3: each box except for the `Node` boxes is a subclass of the `Connector` base class.

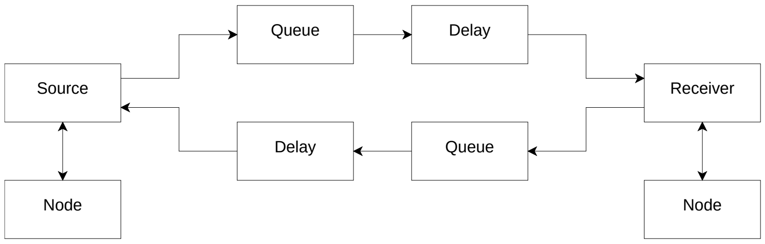


Figure 2.3: An ns-2 Network topology

This simple structure served its purpose really well but when more complex link-level models were introduced to simulate wireless and satellite links, the lack of more explicit interfaces between each layer lead to major incompatibilities between these models as each separate developer relied on different assumptions about the way the system was assembled. For example, this lead to complex workarounds to enable simulation scenarios that use wired, wireless, and satellite models together to ensure that packet routing would work across network boundaries.

2.5.2 OMNeT++

The component model that is used in OMNeT++ is very similar to the ns-2 `Connector` mechanism: each `cComponent` exports a set of input and output `cGate`s. The output gates can be connected to the input gates of other components through `cChannel`s which allow arbitrary `cMessage` objects to be exchanged between the `cComponent`s they connect. To communicate between two modules, one needs to create a message of the right kind and send it on an output gate assuming that the module that is connected on the other side of the output gate knows how to handle the incoming message. In practice, though, some of the components that make up a simulation (for example, the per-host `InterfaceTable` component that maintains the list of network interfaces that exist within a node) do not define any input or output gates: the other components that are present in the simulation access them directly through function calls without sending messages.

While this architecture is very convenient to enable distributed simulations because the simulation runtime has explicit information about the set of objects that can send messages to each other and the delays that each connection applies on these messages, it can be sometimes hard to understand how to connect existing components together and how to replace some of them with new implementations that provide new functionality. The issue here is that there is no mention in the component definition of the set of messages that the component is committed to handle: the message types are usually defined separately and it takes quite a bit of experience to understand exactly which messages a given component can receive or send and the semantics associated with these messages.

2.5.3 GTNetS

To avoid the lack of a clear contract introduced by an abstract simulation model such as the ns-2 model or the OMNeT++ component model, GTNetS focuses on providing a much more explicit separation between each protocol layer. It thus defines the `L2Protocol`, `L3Protocol`, and `L4Protocol` classes that specify how packets can be sent up and down the protocol stack across each layer and that must be subclassed to add new compatible protocols. More importantly, though, GTNetS also defines the `Interface` base class that abstracts a typical network card under a common interface that is used by the higher IP and ARP layers.

2.5.4 ns-3

The modeling philosophy that was chosen by ns-3 is very close to the GTNetS abstractions and proceeds from a number of considerations on how real-world network stacks are built.

In the real world, only two major protocol interface abstractions are in wide use: the most obvious one is the *socket* programming interface that defines how user space applications can send and receive traffic over layer 2, 3, or 4 protocols. The second abstraction is less widely seen by application developers but crucially important for network simulators: it is the kernel-level interface that defines how the layer 3 protocols and the network cards present in the host communicate. This abstraction is defined by the very similar `net_device` and `ifnet` data structures on Linux and BSD respectively.

For efficiency and performance reasons, the other protocol interfaces have not yet converged to a simple set of common features. For example, the interface between the UDP and the IP layer exposes a lot of details of the IP layer to optimize the case where IP datagrams need to be fragmented to make sure that the UDP layer does not allocate a big buffer to hold the user datagram only to create later a set of smaller fragments in the IP layer.

ns-3 adopts a similar structure: most inter-layer interfaces except for the *socket* and `net_device` abstractions are left unspecified to leave as much flexibility as possible to model developers, yet make it easy to inter operate with protocol implementations that expect a *socket* or a `net_device` interference to be present.

- The `NetDevice` class is based on the Linux kernel-level `net_device` data structure: it represents a single hardware device and allows the upper layers to queue packets for transmission by the hardware and makes it possible for the hardware device to notify the upper layers that a new packet has been received.

- `Channel`s interconnect `NetDevices`: each `Channel` object contains the list of `NetDevice` objects it is connected to. It represents a network cable, a wireless transmission medium, or a fiber cable.

- The `Node` class matches the GTNetS concept of a host system. A `Node` contains a list of `NetDevice` and `Application` objects

- The `Socket` class represents a communication endpoint that can be created by applications and that are used to send and receive traffic to the protocol stacks that are attached to a node.

- The `SocketFactory` class can be used to create sockets of various kinds by applications: each protocol that is attached (aggregated) to a node needs to also aggregate a new type of `SocketFactory` to the node so that applications interested in sending or receiving traffic over this new protocol can request this socket factory and create sockets from it.

Figure 2.4 summarizes the relationship between these objects and illustrates how a simple network simulation could be put together to simulate a single link interconnecting a node that sends traffic with a node that receives it.

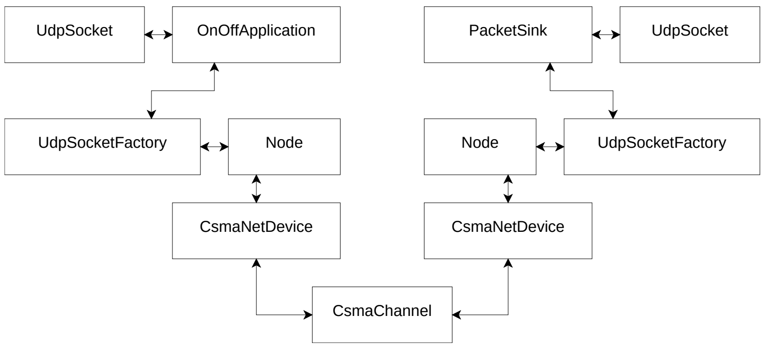


Figure 2.4: An ns-3 Network topology

2.6 Conclusion

In this chapter, we discussed the goals that led us to design a new network simulator and outlined how these goals shaped the subsequent management and technical development of ns-3.

Rather than build ns-3 within a small research team, we have adopted a GPL license with an open development process to ensure that a healthy and active contributor community takes over the maintenance burden from the original ns-3 developers when they move on to other projects.

Our concerns about ease of use and robustness led us to adopt C++ templates and 128-bit integer arithmetic within our time management facility while our need for a high degree of model realism drove the design of the ns-3 network models towards an architecture that is very close to the architecture of a real-world UNIX system.

In the following chapters, we build upon the foundations presented here to demonstrate how ns-3 can be used to bridge the gap between traditional network simulations and field experiments. In chapter 3, we first focus on the design and implementation of the critical component needed to allow realtime ns-3 simulations to interact transparently and efficiently with the real world. Then, in chapter 4, we detail the construction of a simulation environment that can directly execute both user space and kernel space network protocol implementations.

Bibliography

[1] A garbage collector for C and C++. URL `http://www.hpl.hp.com/personal/Hans_Boehm/gc/`. (Accessed September 5th 2010). 28

[2] Cairo. URL `http://cairographics.org/`. (Accessed September 5th 2010). 19

[3] The Network Simulator NS-2. URL `http://www.isi.edu/nsnam/ns/`. (Accessed September 5th 2010). 13, 16, 24

[4] *Optimized link state routing protocol for ad hoc networks*, August 2002. 36

[5] The Pcap file format. URL `http://wiki.wireshark.org/Development/LibpcapFileFormat`. (Accessed September 5th 2010). 35

[6] Cross Platform COM: XPCOM. URL `https://developer.mozilla.org/en/xpcom`. (Accessed September 5th 2010). 30

[7] Andrei Alexandrescu. *Modern C++ design: generic programming and design patterns applied*. Addison-Wesley, 2001. 27

[8] Nicola Baldo and Marco Miozzo. Spectrum-aware channel and PHY layer modeling for ns3. In *VALUETOOLS '09: Proceedings of the Fourth International ICST Conference on Performance Evaluation Methodologies and Tools*, pages 1–8, ICST, Brussels, Belgium, Belgium, 2009. ICST. 36

[9] Rimon Barr, Zygmunt J. Haas, and Robbert van Renesse. JiST: an efficient approach to simulation using virtual machines. *Softw. Pract. Exper.*, 35(6):539–576, 2005. 12

[10] Don Box. *Essential COM*. Addison-Wesley, 1998. 30

[11] J.H. Cowie, D.M. Nicol, and A.T. Ogielski. Modeling the global Internet. *Computing in Science Engineering*, 1(1):42 –50, Jan 1999. 12

[12] Ralph Johnson John M. Vlissides Erich Gamma, Richard Helm. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. 28

[13] Jahanzeb Farooq and Thierry Turletti. An IEEE 802.16 WiMAX module for the NS-3 simulator. In *Proceedings of the SIMUTools Conference*. ICST, 2009. 36

[14] Karl Fogel. *Producing Open Source Software*. O'Reilly, 2005. 14

[15] Richard M. Fujimoto, Kalyan Perumalla, Alfred Park, Hao Wu, Mostafa H. Ammar, and George F. Riley. Large-Scale Network Simulation: How Big? How Fast? *Proceedings of the International Symposium on Modeling, Analysis, and Simulation of Computer Systems*, 0:116, 2003. 16, 24, 28

[16] OPNET Technologies Inc. OPNET Modeler. URL `http://www.opnet.com/`. (Accessed September 5th 2010). 13

[17] Mohamed Amine Ismail, Giuseppe Piro, Luigi Alfredo Grieco, and Thierry Turletti. An Improved IEEE 802.16 WiMAX Module for the ns-3 Simulator. In *Proceedings of SIMUTools Conference*. ICST, 2010. 36

[18] Sam Jansen and Anthony McGregor. Simulation with real world network stacks. In *WSC '05: Proceedings of the 37th Winter Simulation Conference*, pages 2454–2463. Winter Simulation Conference, 2005. 36

[19] Jaroslav Kačer. Discrete event simulations with J-Sim. In *PPPJ '02/IRE '02: Proceedings of the inaugural conference on the Principles and Practice of programming, 2002 and Proceedings of the second workshop on Intermediate representation engineering for virtual machines, 2002*, pages 13–18, Maynooth, County Kildare, Ireland, 2002. National University of Ireland. 12

[20] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*. SpringerVerlag, 1997. 33

[21] Mathieu Lacage and Thomas R. Henderson. Yet another network simulator. In *WNS2 '06: Proceedings of the 2006 workshop on ns-2: the IP network simulator*, page 12. ACM, 2006. 13, 28, 36

[22] IEEE Task P754. *IEEE 754-2008, Standard for Floating-Point Arithmetic*. August 2008. URL `http://ieeexplore.ieee.org/servlet/opac?punumber=4610933;` `http://en.wikipedia.org/wiki/IEEE_754-2008`. 16

[23] C.E. Perkins and E.M. Royer. Ad-hoc on-demand distance vector routing. pages 90 –100, feb. 1999. 36

[24] G.F. Riley, M.H. Ammar, and E.W. Zegura. Efficient routing using NIx-Vectors. pages 390–395, 2001. 36

[25] Sally Floyd Steven McCanne and Kevin Fall. The LBNL Network Simulator. URL `http://ee.lbl.gov/ns/`. (Accessed September 5th 2010). 13

[26] Andrew S. Tanenbaum. *Computer Networks, 4th ed.* Prentice Hall, 2002. 23

[27] Scalable Network Technologies. QualNet. URL `http://www.scalable-networks.com/products/qualnet/`. (Accessed September 5th 2010). 13

[28] András Varga. The OMNeT++ Discrete Event Simulation System. *Proceedings of the European Simulation Multiconference (ESM'2001)*, June 2001. 13, 16, 17, 24

[29] Xiang Zeng, Rajive Bagrodia, and Mario Gerla. GloMoSim: A Library for Parallel Simulation of Large-scale Wireless Networks. In *Proceedings of the Workshop on Parallel and Distributed Simulation*, pages 154–161, 1998. 17

Chapter 3

Simulated Packets

While a great deal of time was invested in the design and the implementation of the ns-3 simulation core, even more effort was invested in the ns-3 network-specific models to fulfill our long-term goal of building a simulation platform which can be used to easily switch back and forth between simulations, testbeds, and field experiments.

In this chapter, we consider how our focus to support natively transparent emulation and real code execution shaped the design and the implementation of the ns-3 simulation packets. We first discuss in 3.1 the requirements we set out to deal with. We then outline in 3.3 the implementation solutions chosen in other simulators and summarize in 3.4 the set of usage patterns we gathered from these existing simulators to guide our implementation. 3.5 describes the details of the implementation we eventually came up with and 3.6 shows the performance impact of these choices on typical simulation scenarios.

3.1 Requirements

When the ns-3 project started, the design of simulation packets spurred many heated and contentious discussions: the reason for this became clear only later but it should have been obvious that the list of requirements we wanted to fulfill was daunting. First and foremost, we needed a number of important features:

- **Support fragmentation and reassembly**: both the IP and the MAC layers often perform these operations and they have a major impact on the behavior and performance of network protocols but many simulators do not provide any support for them.
- **Transparent real bytes**: emulation, real-world code integration in the simulator but also the generation of PCAP [2] trace files require simulation packets to be converted back and forth into real network packets with their native byte encoding. Ideally, this conversion should be fully automatic and transparent to model developers.
- **Simulation data**: it is very common to store extra simulation-only information in some simulated packets to simplify the implementation of some protocol models or to make it easy to test a new idea. For example, to measure the one-way link delay, we can store in a packet the time at which it is sent, and calculate in the receiver the difference between the current receiving time and the send time stored in the packet.
- **Pretty printing**: for debugging or tracing purposes, it is very convenient to be able to generate automatically a human-readable textual representation of the payload and the protocol headers and trailers of a packet.

We also wanted to make sure that the resulting system would be highly efficient both CPU-wise and memory-wise because we knew from experience that the performance of a network simulator critically depends on its ability to perform per-packet processing:

- **Memory efficiency**: when users don't care about the exact content of the application-level payload, it should be possible to solely keep track of its size without allocating the corresponding memory. The simulation of protocols such as high-speed TCP over high-bandwidth links is a typical use-case where this optimization is necessary to avoid saturating the memory of the simulation host.

- **Runtime efficiency**: it is critical that packet management does not become a CPU bottleneck to avoid slowing down every simulation.

Finally, we cared about the overall soundness of our programming interfaces: the long-term success of the simulator depends on its performance but also more importantly on a robust programming interface:

- **Extensibility**: it should be possible to add new types of packet headers or footers without having to modify a shared header so that users who extend the simulator with new network models don't have to go through painful merge sessions to make their models work together.

- **Memory management**: it should be easy to manage packets and to make sure that they get destroyed when they are not needed. Simulations make this problem especially acute because a single packet can be shared by many different protocol layers within a single or multiple network nodes.

- **Hard-to-abuse programming interface**: the need to provide a very efficient data structure should be balanced against the need to ensure that the resulting programming interface is both easy to use and hard to abuse. Efficient implementations often leak out implementation details to their users by using optimized non-transparent memory management rules or awkward programming interfaces.

3.2 Definitions

To avoid ambiguity in the following sections, we define here the primitive data structures used throughput this chapter and then the operations which manipulate them.

Data structures

Header or **Footer**: the fields of this data structure describe the content of a message that is being sent or received by a specific protocol. Each protocol specifies the semantics and the encoding that must be used to serialize these fields in a stream of bytes before sending them over a network. There is no difference between a header and a footer except for their position in the stream of bytes. In the rest of this chapter, we ignore footers.
Packet: an object which contains an ordered list of protocol headers that will be sent from first to last on a network.

Operations

AddHeader is a common operation that is performed on a packet to insert a new protocol header at the head of the list of protocol headers.
RemoveHeader removes and returns the protocol header that is located at the head of the list of protocol headers.
PeekHeader: returns but does not remove the protocol header that is located at the head of the list of protocol headers.

3.3 Related work

The seemingly infinite list of requirements we set out to tackle did discourage us initially because there was no obvious way to chose which requirement we should attempt to address first. Luckily, there was a lot of material to learn from in the form of existing implementations in other simulators. We thus selected a few well known as well as other less well known simulators and proceeded to analyze how their packet facility is designed.

3.3.1 ns-2

The `Packet` data structure implemented in ns-2 [1] is an un-ordered aggregate of protocol headers: it contains a single byte buffer where only one header instance of each registered header type can be stored. Figure 3.1 illustrates the memory layout of this class when only a few types of headers are registered. The headers actually used are shown in light gray.

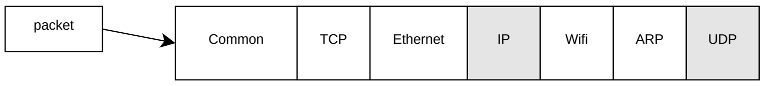


Figure 3.1: An ns2 packet

By default, the set of registered header types is equal to the list of all existing header types which makes ns-2 packets use and waste considerable amounts of memory as each packet instance uses only a very small subset of the total set of registered headers. It is possible to minimize this per-packet memory overhead by specifying a smaller list of registered headers, but very few users know about this functionality or make use of it.

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, an integer used to uniquely identify each packet instance, on-the-side information for various routing protocols, but also a lot of very random and dubious fields 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 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 approach suffers from several weaknesses:

- Every new protocol header must be declared in the shared `packet.h` source file header and the shared tracing classes must be independently modified to include the appropriate pretty-printing code. This systematically introduces software merge problems when the time comes to use together two models designed and implemented separately.
- It is not possible to store two instances of the same type of header in the same packet which makes models such as IP over IP very hard to implement.
- There is no support for fragmentation and reassembly: fragmentation must be implemented by every user by creating a new fragment packet which holds a reference to the original packet and reassembly must get back the original un-fragmented packet.
- There is no way to know the type of and to pretty-print the headers actually present and used in a packet beyond the last header that was added to it.
- Emulation and pcap trace file generation, which both need to convert these simulation packets into real network packets, require an external conversion function which knows precisely the set of headers present in a packet and which knows how to map them back and forth to real network packets. This conversion function must be coded by hand for each combination of the protocol headers that are expected to be found in the packet.
- Packet ownership must be carefully managed to avoid memory leaks, or, worse, double frees.

3.3.2 GTNetS

GTNetS [3] packets represent a considerable improvement over ns-2 packets: each protocol header is represented as a subclass of the `PDU`[1] base class and must provide serialization and de-serialization 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 in and popped out of `Packet` instances through their `PushPDU` and `PopPDU` methods. When a PDU is pushed, the packet takes ownership of the PDU and never releases it. When a PDU is popped, the packet returns a pointer to the PDU stored

[1]This class name reuses the acronym PDU which stands for Protocol Data Unit. Note that this use of the term PDU to refer to a protocol header data only is not compatible with the widely-accepted definition of this term in telecommunication systems where it usually refers to a protocol header and the associated payload.

internally. This means that its users do not need to free the PDUs returned by a call to `Packet::PopPDU`. The stack of `PDU` instance pointers is shown in figure 3.2: this diagram also shows how GTNetS is careful to reserve sufficient room in its buffer of pointers to avoid costly buffer resizes during push in most cases.

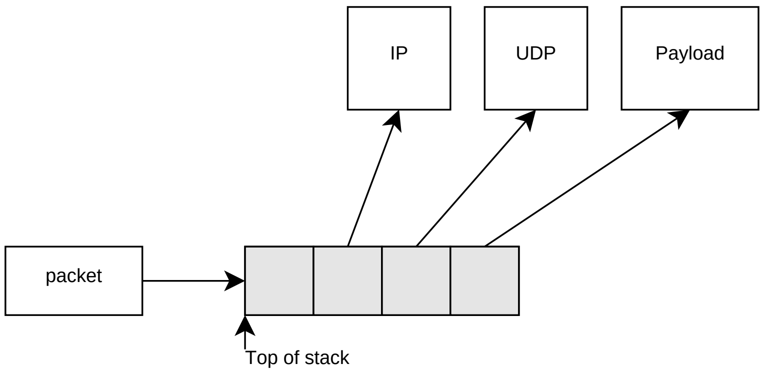


Figure 3.2: A GTNetS packet

This design solves many of the problems identified in the previous section:

- It is very easy to add, completely independently from anyone else, 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 such as IP over IP
- The content of each packet can be entirely printed for debugging or tracing purposes with the `DBPrint` method which loops through the PDU stack to invoke the `Trace` method on each PDU.

However, a number of issues are still unresolved: there is still no provision for fragmentation. Ownership management requires the use of `new` and `delete` which makes it hard to keep track of which piece of code is responsible for deleting a packet created somewhere else. There is also no support for conversion to and from real-world network packets. Furthermore, while this implementation achieves excellent performance (see 3.6), it does so by providing a programming interface that is inherently fragile. For example, `PopPdu` returns a pointer to a PDU instance whose lifetime is managed by the packet class: users have no way to know how long the pointer they obtained will remain valid which can lead to very hard to debug memory corruptions and crashes.

3.3.3 OMNeT++

OMNeT++ [5] packets are very similar to GTNetS packets which means that they share both their strengths and their weaknesses. Each protocol header is modeled as a subclass of the `cPacket` base class: it contains a pointer to the packet it encapsulates, hence forming a singly-linked list of encapsulating packets. This singly-linked list shown in figure 3.3 is logically equivalent to the GTNetS stack of `PDU`s.

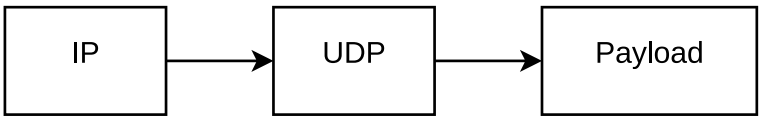


Figure 3.3: An OMNeT++ packet

The programming interface offered by these OMNeT++ packets avoids the subtle pitfalls in which the GTNetS implementation fell: although the `cPacket` class still transfers pointer ownership across its public functions, they do not imply the ambiguous lifetimes offered by the GTNetS `PopPDU` and `PushPDU` methods. Furthermore, the memory management of these packets is more coherent and although it does not completely automate everything, the OMNeT++ component system alleviates a lot of the manual labor needed to manage packets and their headers.

One should also note that the `dup` method that creates a logical deep copy of a packet implements in fact a form of Copy-On-Access that avoids a lot of header copies. Figure 3.4 illustrates the behavior of this mechanism: when the original packet is `dup`ed, OMNeT++ creates a new IP2 header that shares the UDP1 header and it's only if and when the user attempts to call the `decapsulate` function to separate UDP1 from IP2 that UDP2 is unshared from UDP1 on the fly.

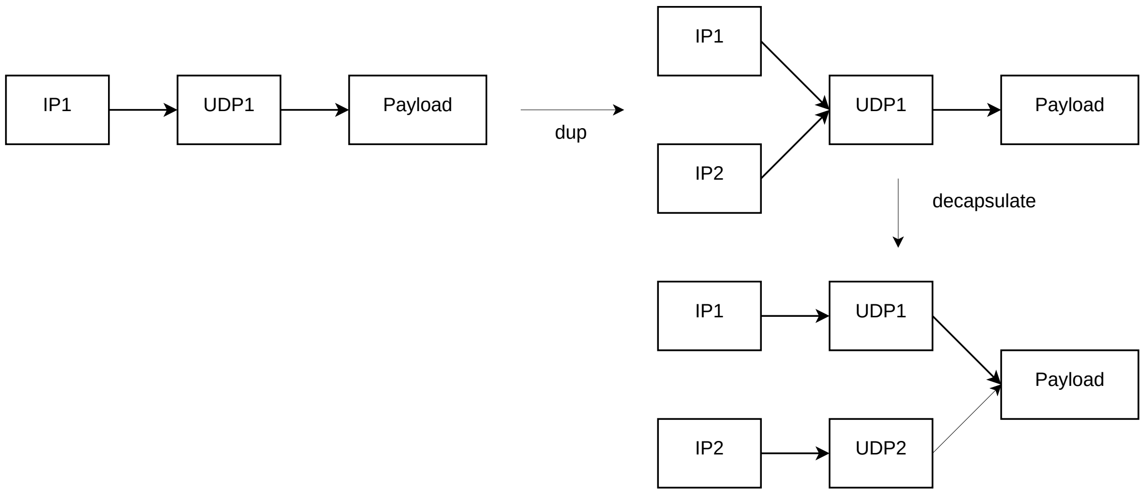


Figure 3.4: `dup` shares headers. `decapsulate` unshares them.

3.3.4 Yans

The approach chosen in Yans [4] is radically different from the other implementations considered until now. Its first concern is to make sure that its programming interface is as robust as possible: it thus avoids transferring pointer ownership across its public methods. For example, to extract a header from a packet, GTNetS does something like this:

```
Packet *p = ...;
IPV4Header *ipv4 = new IPV4Header ();
p->PushPDU (ipv4);
ipv4 = p->PopPDU ();
```

which transfers ownership of the pointer from the caller to the callee in the `PushPDU` method. This memory management policy requires that a user who creates the header must remember to never delete it once it is pushed in a packet to avoid the memory corruption triggered by a double-free. The OMNeT++ `encapsulate` and `decapsulate` methods are similar in design and suffer from a similar problem.

Yans, on the other hand, uses the following pseudo-code which copies the header data structures back and forth between the caller and the callee to make sure that there is no ambiguity about the ownership of this object: whoever creates it must also delete it.

```
Packet *p = ...;
Ipv4Header ipv4;
p->AddHeader (&ipv4);
p->RemoveHeader (&ipv4);
```

Another common source of packet memory management problems comes from the fact that packets are always passed around by pointer for efficiency reasons: it is easy for multiple pieces of code located in different models to hold a pointer to the same underlying packet. Consequently, it is hard for these models to figure out which of them is allowed to delete the packet and more importantly when it is allowed to do so.

GTNetS provides no special facility to deal with this problem which might be fine when a simulator is developed by a small group of tightly knit people but which is unlikely to scale to a large system built by independent developers. OMNeT++ alleviates these problems by automatically keeping track of the ownership of each packet through its component system. To save its users from having to coerce their model implementations within the constraints inherent to a component model such as the one used by OMNeT++, Yans uses a simpler `PacketPtr` reference-counting smart pointer to automatically allow packets to be deleted when no one references them anymore.

In Yans, protocol headers are represented by subclasses of the base class `Chunk`. Each such subclass must implement three methods: `add_to`, `peek_from` and `remove_from`. `add_to` is expected to reserve enough space in the packet byte buffer and to serialize its data into the reserved space. `peek_from` must de-serialize its data from the packet 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. To avoid repeated memory re-allocations due to insertions at the head or the tail of the buffer, all three programming interfaces manipulate a byte buffer initially created sufficiently big to hold all of the protocol headers expected to be inserted before the packet is finalized. Furthermore, to avoid having to move around memory in their protocol buffer, they also start writing protocol headers in the middle of this byte buffer. Figure 3.5 shows what this data structure looks like in memory after inserting a UDP header in front of the application payload: there is still room in front of the buffer to insert the missing IP and Ethernet headers.

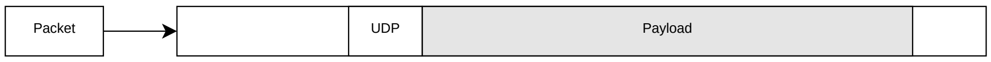


Figure 3.5: The Yans protocol byte buffer

The major difference between the Yans implementation and the Linux and BSD implementations is that, usually, the kernel hardcodes the magic numbers used to reserve enough space at the start of the buffer when it is created. Yans, on the other hand, dynamically adjusts the initial space of each new buffer created based on the past total number of buffer resizes and memory moves it needs to perform while running the simulation.

The serialized representation of each header in the protocol byte buffer must be the *exact* network representation of the header. The in-memory representation of a packet becomes 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`: this facility compensates for the fact that if the chunks have to serialize and de-serialize only their exact network representation, there is no way to store simulation-specific information in each packet.

Contrasting with the other options considered so far, this design makes it trivial to support packet fragmentation, reassembly, and re-packetization since a simulation fragment maps naturally to a network fragment. The `Packet::copy (start, end)` method can create fragments and they can be concatenated with the `Packet::add_at_start` and `Packet::add_at_end` methods. Emulation and pcap trace file generation similarly become completely transparent.

Finally, one should note that while this approach solves some problems, it also suffers from numerous drawbacks. For example, it offers no way to pretty-print automatically the content of a packet, its memory footprint is very high since it always includes the application-level payload, even when its content does not matter, and it is not very efficient CPU-wise.

3.3.5 GloMoSim

GloMoSim [6] packets use an approach similar to Yans'. The protocol byte buffer is allocated initially big enough to hold all the payload and headers which are later copied in it. This allows GloMoSim to later avoid a costly buffer resize through reallocation and copy. GloMoSim also assumes that the size of each header in the protocol byte buffer matches the size of the protocol header being simulated. For example, 1000 bytes of application-level payload are represented by reserving 1000 bytes in the protocol byte buffer. While the main reason why this was done in Yans was to support transparent emulation, it is not obvious why this was done here. Transparent emulation might have been a good reason but the content of the simulated protocol headers does not always match the content of the real protocol header. Simplicity of implementation might be the actual reason for this buffer allocation strategy since a unified approach to handling both application payload and protocol headers considerably decreases the complexity of the system.

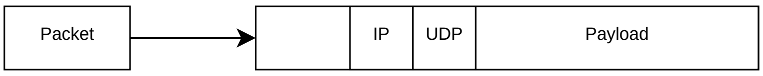


Figure 3.6: A GlomoSim protocol byte buffer

The similarity between the Yans and the GloMoSim implementations implies that they share the same weaknesses: GloMoSim provides no way to automatically pretty-print the content of a packet for debugging purposes and suffers from an inefficient memory footprint when the content of an application payload does not matter but still needs to be allocated.

3.3.6 Summary

Table 3.1 provides a synthetic overview of the set of features discussed in this section and estimates the complexity of the underlying implementations by reporting their size in Kilo Lines of Code (KLOC). While the size of their implementations are all within the same range, this table highlights how their different implementation decisions lead to very different feature sets.

On the one hand, we can see that GTNetS and OMNeT++ which implement their packet facility with a list of protocol headers both support automatic pretty printing, but do not know how to transparently convert their simulation packets to and from network byte buffers and do not provide any simple way to fragment and re-assemble packets.

On the other hand, Yans and GloMoSim store their protocol headers in a buffer of bytes which makes it hard for them to provide a pretty-printing facility but allows them to easily support network byte buffers transparently and to export their functionality through a robust programming interface that does not transfer ownership of pointers between the caller and the callee.

While these two implementation choices might seem exclusive, the ns-3 implementation demonstrates in later sections that it is possible to reconcile them and to obtain a feature set that is the union of the feature sets of these two approaches but that doing so results in considerably higher implementation complexity.

3.4 Usage patterns

The first prototypes of the ns-3 packet facility were implemented without much formal thinking, but we eventually had to gather a set of reference usage patterns against which we could judge our implementation. In this section, we present a number of usage patterns of packet header encapsulation and de-capsulation which we identified through reviews of models in other network simulators. The pseudo-code that we present here is always based

	GTNetS	OMNeT++	Yans	GloMoSim	ns-2
Code size (KLOCs)	1.4	1.6	1	0.8	0.9
Pretty-printing	✓	✓			
Transparent real bytes			✓	✓	
Fragmentation			✓		
Reassembly			✓		
Memory efficiency	✓	✓			
Extensibility	✓	✓	✓	✓	
Robust API			✓	✓	
Simulation data	✓	✓	✓		✓

Table 3.1: Features found in existing simulator packet implementations

on UDP and IPv4 for simplicity but the names UDP and IPv4 could be easily changed to others with no loss of generality. For example, the reception pattern presented below still applies if IPv4 is replaced by an IEEE 802.11 MAC layer.

3.4.1 Packet reception

The simplest usage pattern we identified immediately is that of packet reception. When a packet is received within a layer of the network stack, its headers or footers are peeled away and the packet is then forwarded up the stack if needed. The most typical use-case involves one layer forwarding the packet to only one receiver at the upper layer but there are still many cases where there could be more than one receiver at the upper layer. Examples of the former are plentiful: it includes the IP layer sending a packet up to the transport layer, the IEEE 802.11 MAC layer sending a packet up to the network layer, *etc.*

The cases where there are multiple receivers at the higher layer are less common but they do exist: the UDP layer could be forwarding up the same multicast packet to to a set of multicast applications which have joined the same multicast group but it could also be the case that the IP layer is sending a packet up to the transport layer and a special trace hook.

In general, the common theme in this case is that the byte buffer of a packet is never written into: each layer incrementally parses the packet without touching its content and forgets about the packet as soon as it has been forwarded to the upper layers. This is usually implemented as follows:

```
void Receive (Packet *packet)
{
  Ipv4Header ipv4;
  packet->RemoveHeader (ipv4);
  for (uint32_t i = 0; i < m_receivers.size (); i++)
    {
      m_receivers[i]->Receive (packet);
    }
}
```

This code relies on a number of assumptions about the receiving functions:

- If they merely want to look at the next header, they will use the `PeekHeader` operation to make sure that they do not modify the current parsing state of the packet as seen by the other receiver functions.
- If they need to do more work than just look at the next header, they will first `Copy` the packet to make sure that they do not modify the current parsing state of the packet as seen by the other receiver functions, and then only remove the next header or perform other state-modifying operations.

3.4.2 Packet forwarding

Packet forwarding always starts with a reception but one layer in the stack decides that it needs to re-send the same packet to another node in the network. The classic example here is an IP router but Multi Protocol Label Switching (MPLS) forwarders have the same behavior. In general, forwarding happens when no other layer in a network node is interested in the packet which is why naive implementations usually modify in place the packet received before sending it down the stack. However, it is also pretty common for some tracing layers to receive every packet, independently of what is going to happen to this packet and it is hard to predict the relative order in which these tracing and forwarding hooks are called. A more robust implementation thus performs a packet copy before sending the packet back down the stack so that other potential receivers of this packet within the node are not affected by what happens in this function.

```
void Receive (Packet *packet)
{
  Ipv4Header ipv4;
  packet->RemoveHeader (ipv4);
  ...
  Packet *copy = packet->Copy ();
  copy->AddHeader (ipv4);
  Send (copy);
}
```

3.4.3 Packet transmission over a broadcast medium

The MAC layer which is responsible for sending packets over a broadcast medium can generally send unicast as well as multicast and broadcast messages on its transmission medium. When this transmission medium is a broadcast medium (for example, it is a radio medium), every receiver within reception range must de-capsulate the MAC header to test the header's destination address and figure out whether or not the node is expected to receive the packet. This is usually implemented as follows:

```
void Send (Packet *packet, Address dest)
{
  MacHeader mac;
  mac.SetDestination (dest);
  packet->AddHeader (mac);
  for (uint32_t i = 0; i < m_devices.size (); i++)
    {
      if (m_devices[i] == this)
        continue;
      m_devices->Receive (packet->Copy ());
    }
}
void Receive (Packet *packet)
{
  MacHeader mac;
  packet->RemoveHeader (mac);
  if (mac.GetDestination () == m_self)
    ...
}
```

One interesting thing to note here is that in this case, every simulator we looked at hands over a copy of the packet to the receive function of each receiver rather than rely on them to make the copy themselves if needed. While this might be seen as a missed optimization opportunity, we view this instead as a common realization that optimizations that rely on a callee to play by the rules dictated by the caller without any way to check automatically that these rules are obeyed is inherently a bad idea.

3.4.4 Packet retransmissions

Packet transmissions which do not involve a retransmission are straightforward: they usually insert the right header in the packet with `AddHeader` and then send the packet down the stack. When a retransmission is needed, typically because an acknowledgment message was not received, the picture becomes more complex. For example, when the TCP layer sends a packet down the IP stack for the first time, it needs to keep a reference to the packet in case it needs to be retransmitted later and this forces the TCP layer to create a copy of the packet before sending it down the stack:

```
void TcpSend (void)
{
  Packet *packet = m_txBuffer.front ();
  Packet *copy = packet->Copy ();
  TcpHeader tcp;
  copy->AddHeader (tcp);
  m_ipv4->Send (copy);
}
void AckReceived (void)
```

```
{
  Packet *packet = m_txBuffer.pop_front ();
  delete packet;
}
```

3.4.5 Summary

The detailed analysis of how simulation models use the packet programming interface in other simulators leads to one major observation: there is a strong tension between trying to optimize the performance of a model by minimizing the number of packet copies and trying to maximize the ease of understanding the models by increasing the number of packet copies to minimize dependencies between unrelated functions. In the next section, we make use of this to focus our optimization strategy on minimizing the cost of packet copies to be able to use them liberally and thus maximize the readability and ease of maintenance of the ns-3 network models.

3.5 Implementation

When the time came to design and implement the ns-3 simulation packet facility, it became obvious that one of the requirements discussed in 3.1 constrained considerably the solutions we could adopt: fully transparent support for emulation requires functionality to automatically convert real network packets into simulation packets and vice versa.

When simulation packets are based on an approach similar to the GTNetS, or OMNeT++ facilities, converting simulation packets to network packets is usually trivial to automate because there is enough information in the simulator to generically iterate the headers and footers present in a packet and to invoke a per-header function to convert the simulation data structure into network bytes.

The converse operation, though, is much less trivial because it requires parsing the network packet to reconstruct the set of simulation headers. While it's easy to delegate the conversion from network bytes to simulation data structures for each simulation header once they are created, creating the right simulation headers in the first place is much harder. This problem is well known to software programmers who need to implement serialization and de-serialization functionality and the solution is also well known: the serialized stream must contain in front of each item a type identifier which can be used to instantiate the right object when de-serializing the stream. In this case, though, the format of network packets is not uniform: this makes it impossible to use a fully automatic and uniform serialization and de-serialization solution.

The only approach that makes it possible to really automate the conversion back and forth is to adopt in simulation the normal network representation of a packet so that the conversion back and forth is a matter of copying the entire buffer and waiting for the simulation models to read the right headers and footers in the right order. The ns-3 `Packet` design thus started from the Yans codebase to benefit from its robust API, its memory management scheme based on smart pointers, and its protocol byte buffer stored as network bytes. We altered it to address its many shortcomings: poor CPU performance, abysmal memory usage, and missing automatic pretty-printing of packets.

3.5.1 The zero area

The first issue the ns-3 implementation dealt with was the addition of an optional *zero* area for the application-level payload thus called because it is assumed to be filled with zero bytes. Figure 3.7 shows the dramatic decrease in the size of the protocol byte buffer achieved by keeping track of the size of this zero area rather than allocating space for it: the zero area is colored in light grey to denote the fact that it is virtual and no memory is allocated to represent it.

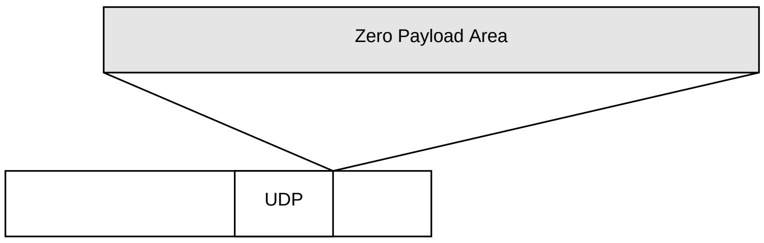


Figure 3.7: The ns-3 byte buffer with a *zero* area.

Although initially implemented solely because of concerns about the memory usage of an ns-3 packet, this memory optimization also turned out to be of considerable interest from a CPU usage perspective. The size of the protocol buffer is minimized which leads to a decreased cost for `Copy` operations because there is less data to copy.

3.5.2 Simple Copy-On-Write

The second objective of the ns-3 implementation was to further improve the CPU and memory efficiency of this class in the common use-cases identified in 3.4 by introducing Copy-On-Write (COW) techniques to make packet copy operations as cheap as possible.

The idea behind COW is to implement the `Packet::Copy` function by returning a new reference to the same object, and to defer the real deep copy of the object to the point where two unrelated users of the same object pointer attempt to modify the shared buffer. Before performing an operation which could potentially change the state seen by another user of the object, we start a deep copy to un-share the shared state and then only complete the requested operation.

This optimization, however, can be a win only when state-changing operations are as rare as possible. The first step to implement COW efficiently thus involves identifying packet state with read-mostly accesses and then separating it from state that is more write-oriented. In the case of a protocol byte buffer, the usage patterns described in 3.4 all point clearly in the same direction: the underlying protocol byte buffer is never modified during receive operations while the current parsing state undergoes constant modifications on both the transmission and reception paths. This leads to a packet data structure such as:

```
class Packet
{
private:
  struct SharedData
  {
    uint32_t count;
    uint32_t size;
    uint8_t *buffer;
  } *m_data;
  uint32_t m_current;
};
```

with a Copy function now responsible for merely increasing the reference count of the shared read-mostly data instead of copying it:

```
Packet *Packet::Copy (void) const
{
  m_data->count++;
  Packet *copy = new Packet ();
  copy->m_current = m_current;
  copy->m_data = m_data;
  return copy;
}
```

The `RemoveHeader` function is safe since it does not modify any shared state:

```
void Packet::RemoveHeader (Header &header);
{
  header.Deserialize (&m_data->buffer[m_current]);
  m_current += header.GetSize ();
}
```

while the `AddHeader` function which writes in the shared buffer a new header needs to be modified to check for the shared state, un-share the shared data structure if needed and then insert the header at the front of the buffer

```
void Packet::AddHeader (const Header &header)
{
  if (IsShared ())
    Unshare ()
  m_current -= header.GetSize ();
  header.Serialize (&m_data->buffer[m_current]);
}
```

The Unshare function decrements the shared reference count and creates a deep copy of the shared data structure:

```
void Packet::Unshare (void)
{
  m_data->count--;
  struct SharedData *copy = new SharedData ();
  copy->count = 1;
  copy->size = m_data->size;
  copy->buffer = new uint8_t [] (copy->size);
  memcpy (copy->buffer, m_data->buffer, copy->size)
  m_data = copy;
}
```

Finally, the traditional test for sharedness is whether or not the reference count is strictly bigger than one:

```
bool Packet::IsShared (void) const
{
  return m_data->count > 1;
}
```

This data structure then makes it possible to implement a very efficient receive path because the `RemoveHeader` function never modifies the shared state and thus never triggers any full deep copy of the data buffer. The following pseudo code becomes both safe and highly-efficient:

```
void Receive (Packet *packet)
{
  Ipv4Header ipv4;
  packet->RemoveHeader (ipv4);
  for (uint32_t i = 0; i < m_receivers.size (); i++)
    {
      Packet *copy = packet->Copy ();
      m_receivers[i]->Receive (copy);
    }
}
```

In this case, using COW decreases so considerably the cost of a Copy operation that it makes it possible to use it liberally where others (see 3.4.1) would have avoided it. This considerably decreases the constraints which every receive function must obey and thus makes it both easier to write these functions and easier to modify and maintain a large set of such receive functions located in independent modules.

3.5.3 Advanced Copy-On-Write

The simple-minded approach described above worked amazingly well in many of our reference usage patterns (see specifically 3.4.1 and 3.4.3) but also failed to avoid deep copies of the packet in the more write-oriented patterns such as 3.4.2 and 3.4.4. To illustrate this failure, we turn to the case of a TCP packet transmission where the TCP layer keeps a reference to the underlying byte buffer before starting the transmission. Figure 3.8 describes the state of our payload+TCP buffer shared by two packets (hence, with a reference count of two) before the IP layer inserts its IP header while figure 3.9 shows how the shared buffer has been unshared by creating a copy of the original buffer and decrementing its reference count by one: each packet now holds a reference to a separate unshared protocol buffer with different contents.

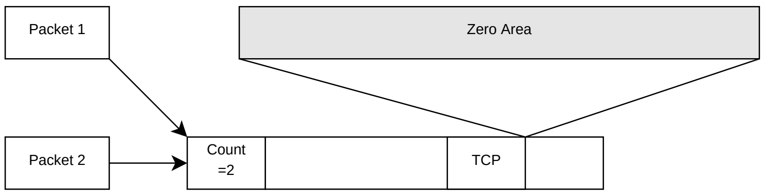


Figure 3.8: The TCP and the IP stacks hold references to a shared buffer.

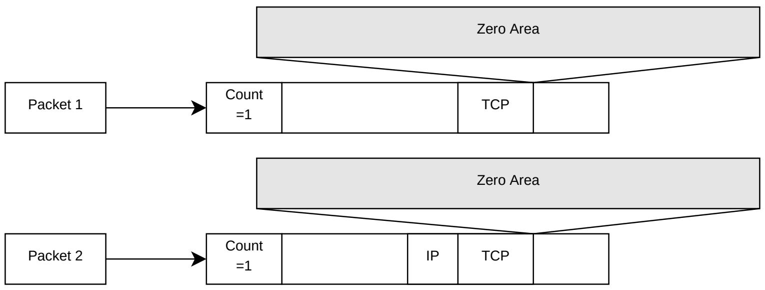


Figure 3.9: The IP stack inserts the IP header, triggers an un-share operation, completes the insertion.

The key to eliminate this deep copy all together is to notice that although both the TCP and IP stack hold a reference to the same packet, they modify different areas of its protocol buffer. A perfect COW implementation would thus be able to track which packets hold a reference to which parts of the protocol byte buffer: this would require that the shared data structure holds a list of back references to the packets it is used by and that each packet remembers which area of the packet it sees.

Figure 3.10 describes these different views of the same underlying data for our TCP and IP packets (for clarity of exposition, it assumes that the zero area does not exist and considers payload to be part of the protocol byte buffer). The *dirty* area referenced by the TCP layer contains solely the application level payload. The *dirty* area referenced by the IP layer contains both the TCP header and the application payload before the insertion of the IP header. After the insertion of the IP header is complete, its *dirty* area is re-sized to also reference the IP header.

If we assume that the shared data structure is extended to contain the list of packets it is referenced by, the introduction of the extra per-packet *dirty* area allows the `IsShared` method to be modified thusly:

```
class Packet
```

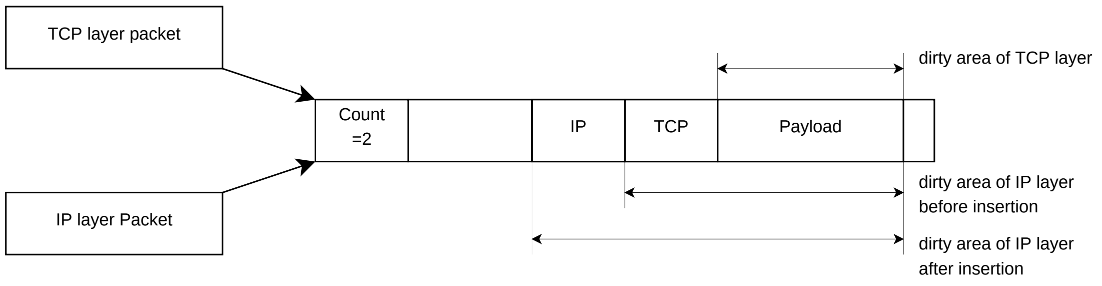

TCP layer packet
IP layer Packet
Count =2
IP
TCP
Payload
dirty area of TCP layer
dirty area of IP layer before insertion
dirty area of IP layer after insertion

Figure 3.10: Keeping track of the *dirty* area

```
{
  ...
  struct SharedData
  {
  ...
  std::vector<Packet *> packets;
  } *m_data;
};
bool Packet::IsShared (void)
{
  if (m_data->count <= 1)
    return false;
  for (int i = 0; i < m_data->packets.size (); i++)
    {
      Packet *other = m_data->packets[i];
      if (other == this)
        continue;
      if (other->dirtyStart < m_dirtyStart
          || other->dirtyEnd > m_dirtyEnd)
        return true;
    }
  return false;
}
```

When the shared reference count is strictly bigger than one, `IsShared` starts a second check to see if anyone who holds a reference to the same shared data also holds a larger dirty area than itself. If no one does, we know that we can safely add new headers or footers outside of our dirty area.

In practice, though, a real implementation does not need the perfect sharing detection provided by the above algorithm and data structures: we can accommodate a certain level of false positives for sharing detection because false positives do not endanger the correctness of our programming interface since they trigger only a couple of extra memory copies.

The sharing detection heuristic implemented in ns-3 is based on a simple observation: it is safe to write outside of the *maximum* dirty area that is the union of all dirty areas of all packets referencing the shared buffer. The ns-3 packet class keeps track of this *maximum* dirty area by updating it whenever a packet writes in some part of the shared buffer:

```
class Packet
{
  ...
  struct SharedData
  {
  ...
    uint32_t dirtyStart;
  } *m_data;
};
void Packet::AddHeader (const Header &header)
{
  if (IsShared ())
    Unshare ()
  m_current -= header.GetSize ();
  m_data->dirtyStart = m_current;
  header.Serialize (&m_data->buffer[m_current]);
}
```

and the IsShared method uses the `dirtyStart` field to try to figure if the user is attempting to write outside (`dirtyStart == m_current`) of the dirty area. If so, there is no need to `Unshare` and `IsShared` can return false:

```
bool Packet::IsShared (void)
{
  if (m_data->count <= 1)
    return false;
  return m_current > m_data->dirtyStart;
}
```

In the case of the TCP/IP example discussed here, this heuristic is sufficient to avoid the deep copy and thus allows us to maximize performance.

3.5.4 Tags: simulation-only per-packet data

Beyond the manipulation of protocol headers and footers, every non-trivial simulation model must deal with other kinds of per-packet information. For example, many users find it convenient to attach the current simulation time to each packet when it is created to be able to infer the one way delay when they receive the packet. Other applications store an integer in each packet to identify the Quality of Service (QoS) class it belongs to so that the local 802.11 stack can enqueue it in the right transmission buffer.

The easy solution for this kind of problem is to add new fields to the `Packet` class. This approach suffers from one major drawback though: over time, it becomes impossible to control the growth of the set of data stored in a packet and this leads to the merging hell alluded to in section 3.3.1 when two developers attempt to use together the modules they developed independently.

Yans introduced the concept of `Tag`s to deal with this issue and ns-3 adopted and extended this solution to allow a user to attach and retrieve an arbitrary blob of data to a packet or to any set of bytes within a packet. This information is stored in on-the-side buffers whose management was implemented with the same COW techniques discussed previously.

Attaching a tag to an ns-3 packet is remarkably similar to the addition of a header to a packet:

```
Ptr<Packet> packet = ...;
MyTag tag;
tag.SetField (...);
packet->AddPacketTag (tag);
```

The implementation of a new kind of tag such as `MyTag` merely requires subclassing the `Tag` base class and implementing the four pure virtual methods it defines.

3.5.5 Metadata: automatic pretty-printing

The choice of a protocol byte buffer which contains solely real network bytes forced us to develop the elaborate `Tag` mechanism discussed above, but it was also the source of one of the most visible missing pieces of functionality: it was not possible to automatically pretty-print the content of a packet because the `Packet` class did not have any information about the semantics of the bytes that it contained. To print the content of a packet for debugging of tracing purposes, ns-3 users had to manually parse the protocol byte buffer which was seen by many as suboptimal.

To support this feature, the ns-3 `Packet` class keeps track in a separate on-the-side data structure of the list of headers and footers that are added or removed explicitly by a user. Each entry in this list describes the type of the header, its size, and the size of the fragment of this header that is present in the protocol byte buffer. When the time comes to print the content of a packet, we parse this list and print information about each header fragment if there are any. Then, for each non-fragmented header, we create a header of a matching type, de-serialize its data from the protocol byte buffer, and invoke its `Print` method.

While this method is very slow, it is never an issue in practice: when a user attempts to pretty-print the content of a packet, he usually does not care about speed because he is debugging or he is dumping the resulting string in a hard disk file which tends to make his simulation IO-bound rather than CPU-bound. The major advantage of this approach is that it is entirely automatic and merely relies on each protocol header to implement a `Print` method.

3.5.6 Summary

In this section, we have given an outline of how the ns-3 packet implementation works and the data structures it uses to provide a robust API that supports the whole range of features we defined as requirements in section 3.1. While fragmentation and reassembly support came naturally from our decision to use a protocol byte buffer that matches an actual network packet, the need to provide automatic pretty-printing of packets and simulation-only per-packet user data forced us to integrate respectively a metadata and a `Tag` mechanism.

3.6 Performance Evaluation

Although it is important for us to provide in ns-3 a packet facility that supports a large set of features, we also know from experience that the performance of the simulator as a whole depends critically on the performance of its packet facility. In this section, we first consider the problem of defining reference micro-benchmarks that are based on the usage patterns described in 3.4 and that are meaningful to the performance of real simulation scenarios. Then, we compare the performance of the ns-3 implementation with that of GTNetS, OMNeT++, and Yans on these reference micro-benchmarks.

3.6.1 Benchmark description

Translating the abstract usage patterns first outlined in 3.4 into concrete benchmarks is more difficult than it appears: each simulator has different programming interfaces with different semantics and wildly different approaches to memory management. The version of each benchmark used by each simulator is thus necessarily different but we believe that they are sufficiently close from a functional perspective to make these comparisons meaningful. To simplify our implementation, the benchmarks that are described here always used UDP and IPv4 headers even when the original usage pattern is more appropriate for TCP or a IEEE 802.11 protocol.

The list of benchmarks described below contains mostly *safe* versions of the usage patterns described in 3.4: the *safe* adjective refers here to the fact that these benchmarks attempt to mimic a protocol stack implementation that is careful to always perform a packet copy when it is necessary to avoid that multiple receivers of a packet do not interfere with each other. Some of the simulators benchmarked here used instead for some of these patterns an unsafe variant which avoided the extra packet copy to optimize their processing paths. These variants are included below and marked with the keyword *unsafe*.

1. **reception**: create a packet which contains payload, UDP, and IPv4 headers, remove the IPv4 header, and send a copy of the packet to one receiver at the higher layer. The receiver removes the UDP header.

2. **forwarding**: create a packet which contains payload, UDP, and IPv4 headers, remove the IPv4 header, copy the packet, insert the IPv4 header again.

3. **transmission**: create a packet which contains payload, insert an IPv4 header, and send a copy of the packet to one receiver. The receiver removes the IPv4 header.

4. **re-transmission**: create a copy of a packet which contains payload, insert UDP and IPv4 headers.

5. **forwarding unsafe**: create a packet which contains payload, UDP, and IPv4 headers, remove the IPv4 header, and insert the IPv4 header again.

6. **reception unsafe**: create a packet which contains payload, UDP, and IPv4 headers, remove the IPv4 header, and then send the same packet to one receiver at the higher layer. The receiver looks at the UDP header without removing it.

3.6.2 Benchmark results

These benchmarks were run on three different systems, including one 32bit i386 system, one 64-bit x86_64 Intel system, and one 64-bit x86_64 Amd system. All results were similar which is why we include only the results from the x86_64 Intel system. Each benchmark ran its iteration loop one million times. They were run at least 10 times, and until the standard relative deviation of their measured execution time drops below 0.05. We recorded the minimum execution time over these runs and report them in table 3.2. Table 3.3 summarizes the relative speedup provided by the ns-3 implementation. The ns-3 metadata column indicates the result of these benchmarks on ns-3 when we enable recording metadata about the set of headers and footers present in a packet to be able to pretty-print its content automatically.

	ns-3	ns-3 metadata	OMNeT++	GTNetS	Yans
reception-safe	0.31	0.35	0.62	0.70	0.97
transmission-safe	0.44	0.50	0.94	0.79	0.95
forwarding-safe	0.42	0.47	1.18	0.80	1.02
retransmission	0.40	0.46	0.77	0.68	1.90
reception-unsafe	0.20	0.23	0.37	0.36	
forwarding-unsafe	0.29	0.34	0.38	0.35	0.62

Table 3.2: Minimum execution time (s)

3.6.3 Summary

Despite the numerous features provided by the ns-3 packet implementation, our benchmarks show that it compares favorably in terms of cpu efficiency against other simulators. Whether metadata recording is enabled or not, the ns-3 packet implementation is consistently the fastest and is up to 2 times faster than the GTNetS and OMNeT++ simulators.

	ns-3	ns-3 metadata	OMNeT++	GTNetS	Yans
reception-safe	1.00	1.11	1.98	2.25	3.07
transmission-safe	1.00	1.14	2.13	1.78	2.16
forwarding-safe	1.00	1.12	2.83	1.92	2.44
retransmission	1.00	1.15	1.92	1.70	4.76
reception-unsafe	1.00	1.15	1.84	1.76	
forwarding-unsafe	1.00	1.18	1.30	1.22	2.12

Table 3.3: Relative speedup

3.7 Conclusion

In this chapter, we demonstrated that it is possible to support efficiently the large set of requirements described in 3.1 and successfully merge together the two major implementation approaches that had been chosen until now and that were championed by GTNetS/OMNeT++ on one side and Yans/GloMoSim on the other side.

Table 3.4 extends table 3.1 to include the ns-3 feature set and illustrates the unavoidable cost of supporting together all this functionality: the complexity of the ns-3 implementation measured by its size in KLOCs is considerably higher than that of any other simulator.

	ns-3	GTNetS	OMNeT++	Yans	GloMoSim	ns-2
Code size (KLOCs)	7.8	1.4	1.6	1	0.8	0.9
Pretty-printing	✓	✓	✓			
Transparent real bytes	✓			✓	✓	
Fragmentation	✓			✓		
Reassembly	✓			✓		
Memory efficiency	✓	✓	✓			
Extensibility	✓	✓	✓	✓	✓	
Robust API	✓			✓	✓	
Simulation data	✓	✓	✓	✓		✓

Table 3.4: Features found in ns-3 and other simulator packet implementations

However, the price in implementation complexity that is paid here is more than offset by the ability of this implementation to transparently support the conversion back and forth between simulation packets and actual network bytes since it is the fundamental building block upon which two major features of ns-3 are built:

- The ability to use ns-3 as a real-time emulator that can be connected to network testbeds and field experiments to extend their scalability.
- A simulation environment that can directly execute existing user space and kernel space protocol implementations within ns-3.

While we will not discuss further the former because there is little to it beyond the transparent packet conversion support described in this chapter, the next chapter deals exclusively with the latter.

Bibliography

[1] The Network Simulator NS-2. URL `http://www.isi.edu/nsnam/ns/`. (Accessed September 5th 2010). 46

[2] The Pcap file format. URL `http://wiki.wireshark.org/Development/LibpcapFileFormat`. (Accessed September 5th 2010). 44

[3] Richard M. Fujimoto, Kalyan Perumalla, Alfred Park, Hao Wu, Mostafa H. Ammar, and George F. Riley. Large-Scale Network Simulation: How Big? How Fast? *Proceedings of the International Symposium on Modeling, Analysis, and Simulation of Computer Systems*, 0:116, 2003. 47

[4] Mathieu Lacage and Thomas R. Henderson. Yet another network simulator. In *WNS2 '06: Proceedings of the 2006 workshop on ns-2: the IP network simulator*, page 12. ACM, 2006. 49

[5] András Varga. The OMNeT++ Discrete Event Simulation System. *Proceedings of the European Simulation Multiconference (ESM'2001)*, June 2001. 48

[6] Xiang Zeng, Rajive Bagrodia, and Mario Gerla. GloMoSim: A Library for Parallel Simulation of Large-scale Wireless Networks. In *Proceedings of the Workshop on Parallel and Distributed Simulation*, pages 154–161, 1998. 51

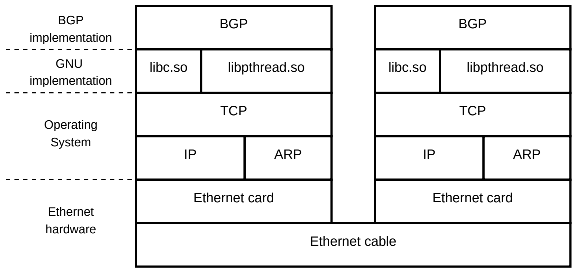
Chapter 4

Direct Code Execution

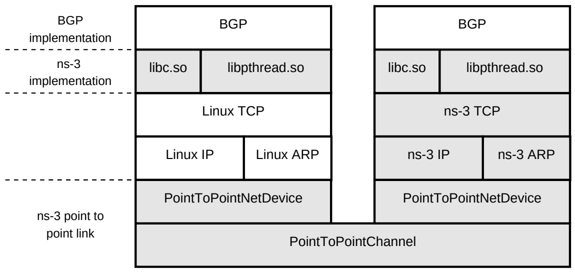
The growing need to be able to conduct realistic experiments which involve complex cross-layer interactions between many layers of the network stack has led network researchers to slowly stop using network simulations because their models are seen as not being sufficiently realistic. One of the objectives of this thesis, however, is to show that experiment realism is not necessarily antithetic to the use of simulators. So far, we have shown that it is possible to seamlessly integrate every model of a simulator in a larger testbed or field experiment by using a real-time simulation scheduler and carefully designing some of the critical components of the simulator such as the packet facility. Such a setup makes it easy to incrementally tradeoff realism for increased scalability in a field or a testbed experiment by interconnecting simulated networks to the system being studied.

In this chapter, we consider the converse operation where a simulator trades off scalability for realism by embedding within ns-3 existing protocol implementations that were never designed to run in a simulator. We thus demonstrate that it is possible to considerably increase the realism of a simulation while retaining its repeatability and ease of debugging and to make it trivial to repeat and compare the result of the same experiment in a testbed by using the same protocol implementation in both cases.

Figures 4.1a and 4.1b illustrate the problem we attempt to deal with: the former describes the case of two Border Gateway Protocol (BGP) routing daemons that are running within UNIX hosts and that are interconnected through a single inter-Internet Service Provider (ISP) link while the latter describes a simulation that directly executes the software components of that system directly within the simulator. The components that are provided directly by ns-3 are highlighted in light grey; the white boxes represent code that was not implemented in ns-3 but that runs within the simulator in simulated time. In this case, the left-most node simulates the BGP daemon on top of the simulated Linux TCP/IP stack while the right-mode node runs the BGP daemon on top of the ns-3 native TCP/IP stack.



(a) Two BGP daemons running on UNIX hosts



(b) Two BGP daemons running within ns-3 with Linux TCP/IP stack

To be able to run the scenario depicted in figure 4.1b, we need to virtualize a number of services so that multiple instances of the same protocol implementation running within our simulator are isolated from each other:

1. First, we need to make sure that each protocol implementation instance does not

share its global and static variables with another instance of the same protocol implementation.

2. Then, to simulate a user application such as the BGP daemon running on top of a UNIX system, we must provide a way to intercept all calls to UNIX-compatible functions and replace them with a simulation implementation. For example, we need to make sure that every call to the UNIX `gettimeofday` function is eventually translated in a call to a function which returns the simulation time instead of the wall-clock time.

3. Finally, to run within the simulator an OS-level protocol implementation such as the Linux TCP/IP stack, we need to intercept calls to OS functions and redirect them to simulation implementations. In the case of the Linux TCP/IP stack, the kernel memory allocation function `kmalloc` is a good example of a function that must be replaced.

While it is quite easy to achieve the above by manually modifying the source code of the relevant protocol implementations and recompiling them, those who have tried to do so have usually regretted it quickly. These large and intrusive modifications are usually similar to what is presented in figure 4.1: global variables are transformed into global arrays of global variables[1] while system functions such as `read` are renamed with a prefix or a suffix such as `sim_read`. When the time comes to re-apply these manual modifications to a new version of the original protocol implementation, most simulator developers simply give up and do not upgrade.

```
static int g_some_var;
void some_function (void)
{
  g_some_var++;
  read (socket, ...);
}
```

```
static int g_some_var[100];
void some_function (void)
{
  g_some_var[current_simulation_id ()]++;
  sim_read (socket, ...);
}
```

Figure 4.1: Manual protocol implementation modifications: `read` is renamed `sim_read` and global variables are transformed into arrays.

The main challenge we deal with in this chapter is thus the problem of building a complete Direct Code Execution (DCE) environment that can execute in a simulator *unmodified* user space and kernel space protocol implementations so that upgrading from a protocol implementation version to another is trivial. [2]

[1] In this example, the `current_simulation_id` function returns a unique id which represents the instance of the protocol currently executing. There are more elaborate variants which make use of macros and dynamic allocation of the global data array to avoid wasting memory needlessly but they are functionally equivalent to this version.

[2] A nice side-effect of this objective is that it should make it possible to run multiple versions of the same protocol implementation within the simulator to test inter-operability problems.

Because many other tools share similar objectives and provide functionality that might appear related, section 4.2 discusses how these existing projects fall short of properly addressing all the requirements we set out to solve.

In section 4.3, we proceed to describe the foundation upon which we build the ns-3 DCE environment: the Virtualizing Dynamic Loader (VDL) Executable and Linkable Format (ELF) loader is used to both virtualize access to every global and static variable and to redirect function calls to system facilities from their normal implementation to a simulation implementation. Later on, in section 4.4 we discuss the binary-compatible re-implementation of the Linux user space libraries we use to run unmodified Linux executables within ns-3 and then we review in 4.5 the kernel space facilities that are re-implemented in ns-3 to run an unmodified recompiled Linux network stack.

Finally, to highlight the CPU and memory usage of the resulting simulation environment, we compare in section 4.6 its behavior against that of the LinuX Containers (LXC) Network Namespace (NetNs) virtualization tool.

4.1 Requirements

In the introduction, we already mentioned some of the important characteristics of a good DCE environment but we state them formally here for clarity:

- **No manual source code modifications**: we must avoid having to perform non-automated source code modifications on existing protocol implementations. The cost of having to regularly port these manual modifications from an old version to a new version of an external protocol implementation is too high to be sustainable in the long term.

- **Integrated debugging**: it should be possible to debug the whole simulation with a single debugger program to be able to easily, inspect, place breakpoints and trace the behavior of every component from a central user interface. Solutions that require the use of a distributed debugger to control multiple processes at the same time are explicitly not acceptable.

- **Efficiency**: the memory and CPU cost of virtualizing a single external protocol implementation for execution within the simulator should be as low as possible to be able to run simulations which process many packets per second and include a large numbers of instances of this protocol implementation.

- **Kernel-space and user-space**: we want to be able to embed within the simulator both kernel-space (the TCP/IP Linux stack for example) and user-space (the BGP Zebra daemon for example) protocol implementations.

- **C and C++**: we need to support at least C and C++ since the protocol implementations we want to reuse are written in both languages.

4.2 Related work

While increasing the realism of a simulation by integrating existing real-world protocol implementations directly in the simulator is not a new idea, none of those who tried to tackle this problem have so far succeeded in providing adequate answers to all the requirements stated above.

4.2.1 GTNetS, ns-2, GloMoSim, Yans

GTNetS, ns-2, GloMoSim, but also Yans have all succumbed to the temptation of integrating manually directly within the source tree of the simulator existing real-world protocol implementations. GTNetS and ns-2 embed [13] the Zebra [6] BGP implementation, GloMoSim [36] uses the Free BSD 2.2.2 TCP stack and Yans [23] stole its TCP stack from Berkeley Software Distribution (BSD) 4.4 Lite.

However, the apparent simplicity and popularity of this approach, should not hide the fact that it is merely a short-term stop-gap solution: the considerable effort needed to perform the initial modification must be repeated over and over again for each subsequent release from the upstream project. The logical consequence is that we are not aware of any simulator which has attempted to keep track regularly of these new releases: the simulator models become stale and their usefulness quickly decreases over time.

4.2.2 Alpine

Alpine [15] was originally developed as a framework to ease the development of new TCP/IP protocols and algorithms: its sole objective was to move the TCP/IP implementation from the kernel to a user-space library. The idea behind this approach is that it alleviates the need to perform systematic system reboots to test each change. Bugs also do not translate anymore in a *blue screen of death*: it is easy to recover from the crash of a user-space library and to analyze it with a simple debugger.

The source code of the FreeBSD 3.3 kernel network stack was thus extracted and recompiled in a shared library together with appropriate wrappers around kernel-level primitives to provide an appropriate runtime environment to the network stack. This shared library also wraps and exports the network stack kernel-level primitives under an implementation of the socket API. Applications which intend to use this TCP/IP stack then merely need to link explicitly against this library to make sure that any call to a socket function is redirected to the Alpine implementation rather than go through the normal kernel path.

Although Alpine requires no manual source code modifications, works for both user-space and kernel-space protocol code, and is reasonably efficient, it is inherently limited to work with a single network stack and a single application per host process. Of course, it would be possible to run multiple instances of Alpine on a single host system to experiment with multiple network stacks but it would not be possible to use more than one application per stack. Furthermore, this would make debugging the resulting set of applications very hard without a distributed debugger since each network stack and application runs in a different process.

4.2.3 nfsim

The netfilter simulation environment (nfsim) [30] was created with objectives very similar to Alpine: the goal was to provide a test framework to automate the testing of the Linux kernel netfilter stack that is used throughout the network stack to intercept and mangle packets for tasks as diverse as connection tracking, dynamic NAT, etc.

These similar objectives translated in similar implementations: only one instance of the network stack can be simulated and normal applications such as `iptables` can be interconnected to the simulator by using an `LD_PRELOAD`ed library providing overriding wrappers for the {`get,set`}`sockopt` functions.

4.2.4 COOJA

COOJA [29] is a network simulation tool that was originally implemented to facilitate the development of protocols for sensor networks using the Contiki [14] OS. The COOJA Java simulation core contains wireless link models that are used to simulate the wireless links interconnecting sensor devices and allows users to execute directly within the simulator binary images of the Contiki OS.

These Contiki images are executed one after the other, one event at a time through JNI bindings. Their data segments are copied back and forth between the Java simulator and the C memory every time a native Contiki event is executed. Unfortunaly, the authors report [29] that this approach to execute multiple native Contiki images within the same process has slightly less than 11% CPU efficiency: more than 89% of the simulation time is spent on bookeeping tasks to merely ensure that each node views a distinct copy of its global and static variables.

4.2.5 Entrapid

Entrapid [18] might be seen as the ancestor of all the tools we discuss here: it predates Alpine by many years and yet achieves a considerably-larger set of features.

To support more than one instance of the network stack within the simulation process, Entrapid integrates a copy of the BSD 4.4 kernel which is manually modified: every declaration of and access to a global variable has been changed as described in section 4.2.1. Entrapid integrates applications by linking them against its local copy of the BSD 4.4 kernel and allows multiple instances of a single application to exist within the simulation process when these applications are re-entrant. Although there is not enough clear information about this, we can safely assume that the default applications shipped with Entrapid were modified manually to remove their global variables and make them re-entrant.

Entrapid used liberally manual source code modifications as a means to integrate external protocol implementations in a network simulator and was thus probably impossible to maintain in the long term (which is also probably the reason for its sudden disappearance). It was nonetheless the first instance of a fully integrated network simulation environment that integrates both user-space and kernel-space protocol implementations within a single process under the control of a single debugger.

4.2.6 Network Simulation Cradle

The Network Simulation Cradle [19] was designed as a simulation tool which could integrate every existing kernel network stack and allow users to compare their behavior easily. Because they wanted to integrate many different external protocol implementations, the project started with the assumption that it would be crucial to avoid source-level modifications of these network stacks to be able to maintain them with few development resources. NSC thus automates completely the set of source-level modifications needed to avoid reentrancy issues and to virtualize the global variables used by each network stack.

The aptly-named globalizer [20] is run through every source file of each network stack to generate a new C file and the resulting C files are then recompiled and linked together into a set of shared libraries which are then used by the network simulator. ns-3 integrated very early on support for NSC as an alternative to its native TCP stack thanks to support from the Google Summer of Code program and we spent a lot of time trying to figure out whether or not this approach could be extended to support the C and C++ applications we had in mind. However, it turns out that, in practice, the source code parser used in [20] is brittle and we eventually realized that parsing source code is inherently a game that we could not win given our limited development resources (see notably [12]).

4.2.7 NCTUns

The approach chosen in NCTUns [34] differs radically from everything we considered so far. Although they started from similar premises, that is, maximize realism by integrating real protocol implementations and minimize effort by minimizing the amount of modifications needed, their simulator departs considerably from other solutions.

NCTUns modifies slightly the source code of its host TCP/IP stack as well as the applications that run in it so that their view of time is synchronized with the external simulation scheduler. Furthermore, they all collaborate to virtualize the host network stack and allow multiple instances of the network stack, each with its own routing table, to co-exist peacefully together. The simulation scheduler is also responsible for modeling the network links interconnecting each network stack.

Small modifications for each protocol implementation coupled with the use of a separate process for each application allows the NCTUns simulator to avoid complex virtualization techniques. The cost, though, is manyfold: the time synchronization algorithm used by the simulation scheduler does not ensure full reproducibility. Furthermore, debugging all the simulated application processes requires a distributed debugger. Finally, the source code modifications needed for applications are not really trivial and require a good understanding of their behavior and their use of time, sockets, and signal-based timers.

4.2.8 IMUNES, OpenVZ, LXC

Similarly to Alpine and in contrast with the other options considered so far, IMUNES [35], OpenVZ [32], and LXC [4] do not use a simulated virtual time. Instead, these tools run experiments in real time. These container virtualization technologies allow users to create within the same host OS kernel multiple instances of its network stack. Contrary to the NCTUns solution which uses an elaborate scheme of virtual IP addresses to virtualize solely the routing table, these tools add a level of indirection for every access to a data structure in the network stack which results in a much more coherent view of the virtualized network stack. Furthermore, they do not require any changes to user-level applications: the user is responsible for creating each virtual network stack and for executing each application within its relevant stack.

However, this container approach still suffers from many drawbacks: first, IMUNES and OpenVZ but not LXC are developed as a large and intrusive patchset on the entire network stack that must be maintained across FreeBSD and Linux releases. Another issue worth mentioning is that these tools cannot be used with a network simulator other than in real-time emulation mode and because each application runs in a separate process, debugging such an experiment can easily become painful without a distributed debugger.

Strictly speaking, neither IMUNES nor its Linux equivalents can be used directly by a network simulator but they are usually seen as the lightest possible virtualization technology both in terms of CPU and memory overhead. Every other virtualization tool based on platform-level virtualization such as [31, 10, 8] introduces more overhead and thus achieves lower CPU and memory performance when under heavy network traffic. Because LXC is the only container approach that has been integrated in the underlying OS kernel, we thus chose to use it in section 4.6, as a baseline against which we can compare the performance of our DCE environment.

4.2.9 dONE and Weaves

The Distributed Open Network Emulator [11] is a network simulator which embeds a copy of the Linux 2.4 network stack [22] as well as UNIX applications such as ping, and ftp. Weaves [28, 27] is used to provide the underlying virtualization technology: its main purpose is to virtualize all access to global and static variables. The approach chosen in Weaves is very similar to NSC: while NSC parses and modifies the source code of the application, Weaves parses and modifies the textual assembly code generated by the compiler. The user is then responsible for assembling this textual assembler file in a binary object file which can then be loaded by the Weaves runtime within the simulator.

Details about the assembly-level modifications performed by Weaves are scarce but it appears to:

- transform every access to a static variable in an access to a global variable so that they go through an extra memory indirection

- assume that the %ebx register is used by the compiler as the base register for all accesses to the global variable indirection table
- remove and replace with ad hoc code the %ebx register setup instructions from the prologue of each function so that the Weaves runtime can control indirectly which indirection table is used for access to global variables

Assuming that %ebx is used as the sole base register for access to global variables appears however especially dubious. As per the i386 Application Binary Interface (ABI) [3], compilers are free to (and do) use any register in leaf functions (functions which do not call other functions) which means that controlling the sole %ebx register is not sufficient to control which global variables are accessed by an arbitrary function.

Sadly, the source code of the assembly parser used by Weaves was never released which makes it hard to verify that the above assumptions are correct. If they are, then the ftp applications used by the Weaves developers to demonstrate their tool either did not contain leaf functions that access global or static variables or were modified manually to not contain any.

However, a more problematic issue is the fact that Weaves relies on register-based addressing to be used to access the indirection table. On many architectures such as 64-bit x86 systems [25], PC-based addressing is instead used by default: rather than access the indirection table through a special register that is setup on entry to the function, the compiler takes advantage of memory access instructions that reference the address of the currently executing instruction stored in the Program Counter (PC). On these systems, the address of the indirection table is directly calculated at each access which makes it impossible to control which indirection table is accessed by merely changing globally the value of a single register such as %ebx.

One can only speculate on the kind of assembly transformation that the authors of this tool intended to implement on such systems but it is likely that they would all have required fairly considerable assembly code analysis and transformation with a level of complexity similar to that of implementing the complete robust C++ parser needed to make NSC work in our use case. This might be the reason for why the Weaves virtualization tool was never released or further developed.

4.2.10 Summary

Because there have been many different approaches to the same or very similar problems, it can be hard to understand the difference and the limitations of each of these tools. Table 4.1 provides a synthetic summary of how each of the discrete event network simulators discussed so far fare against our list of requirements while table 4.2 focuses on the virtualization environments.

The most notable fact to highlight is that there are very few projects that attempted to develop a solution that requires no manual modifications to the protocol stacks, allows multiple nodes to be simulated, and runs in simulated time to make debugging easy. NSC,

	user protocol	kernel protocol	many nodes	easy debugging	automated modifications
GTNetS	✓		✓	✓	
Glomosim	✓		✓	✓	
ns-2	✓		✓	✓	
Yans	✓		✓	✓	
nfsim	✓	✓		✓	✓
COOJA		✓	✓	✓	✓
Entrapid	✓	✓	✓	✓	
NSC		✓	✓	✓	✓
NCTUns	✓	✓	✓		
dOne + Weaves	✓	✓	✓	✓	✓

Table 4.1: Discrete event network simulators: requirements

	user protocol	kernel protocol	many nodes	easy debugging	automated modifications
Alpine	✓	✓		✓	✓
IMUNES	✓	✓	✓		
OpenVZ	✓	✓	✓		
LXC	✓	✓	✓		✓

Table 4.2: Virtualization environments: requirements

dOne/Weaves and COOJA are the only projects that fall in this category. While NSC and COOJA are fairly serious contenders, we are very skeptical that Weaves really worked and, if it worked, that it could have been used on anything but the test applications and the Intel 32bit system used by its authors. NSC, on the other hand has already demonstrated its usefulness and was integrated in ns-3 but its source code parser cannot be extended to support C++ without considerable work. In the following sections, we thus consider only COOJA as a robust albeit slow solution.

4.3 User-space virtualization

Most people might conclude from the review we conducted in previous section that what we are trying to achieve is hopeless and that there is no way to fully automate efficiently the virtualization of the accesses to the global and static variables of a C or C++ program. In this section, we demonstrate however that it is possible to leverage the code generation capabilities of existing C and C++ compilers together with the functionality provided by the system ELF dynamic loader to achieve our objectives.

First, we implement an ad hoc ELF loader that provides excellent robustness and low CPU and memory usage even when large numbers of protocol implementation instances are loaded in memory. We then consider an alternative based on the COOJA virtualization algorithm that takes advantage of the capabilities of the standard ELF dynamic loader found in normal Linux distributions but that exhibits poor CPU and memory usage. Finally, we estimate the relative performance of these two options and highlight the comparatively excellent CPU and memory efficiency of our ad hoc ELF loader.

4.3.1 Background

Before we describe the implementation details of our *User-space virtualization* technique, we start with some background discussion on the structure and the behavior of the code generated by a typical compiler to access global and static variables.

On every CPU/OS combination, the platform Application Binary Interface (ABI) defines precisely the rules that must be followed to access a global or a static variable. Most UNIX systems have converged over time to a common approach usually refered to as the *System V Application Binary Interface* with CPU-specific supplements [25, 3]. All the work discussed here is thus applicable to every modern UNIX OSs and to every CPU. However, for the sake of clarity, all examples will refer to the 32bit Intel CPU architecture.

In general, three methods can be used to access a global or a static variable, depending on which compilation flags were specified during the compilation and link stages,

- direct access by absolute address,
- indirect access by base address, and
- indirect access by current address

The first method does not provide any easy opportunity to virtualize access to global and static variables so we will not discuss it further here.

Indirect access by base address: LTR

```
static int g_a;
int main (int argc, char *argv[])
{
  g_a++;                                mov    0x177c,%eax
                                        add    $0x1,%eax
                                        mov    %eax,0x177c
}
```

If the C source code shown below on the left is compiled with `gcc -c -o test.o test.c`, and then linked with `gcc -pie -o test test.o`, the final executable contains assembly binary code that uses a constant offset (`0x177c` here) from the base address of the code section to the location of the static variable in the data section. In this case, the user also explicitely requested the linker to create a relocation table in the resulting binary (the `-pie` option). The executable thus also contains two `R_386_RELATIVE` relocations in its relocation table that reference the address of the two `mov` instructions used above. When the dynamic loader maps this binary in memory, it patches the memory locations referenced by these relocations by adding to them the base address of the binary.

These Load-Time Relocation (LTR)s rely on the compiler and the loader to collaborate to make sure that the offset between the code and data sections is a constant, regardless of which base address is used to load the code section. Figure 4.2 illustrates this relationship.

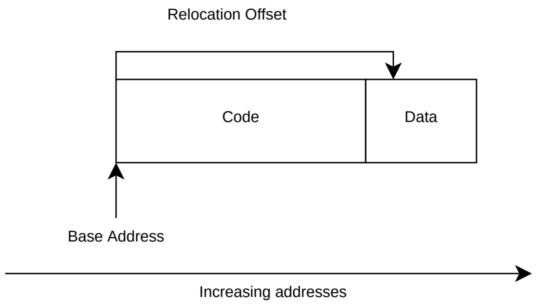


Figure 4.2: Memory layout of a Load-Time Relocatable binary

These rules make it possible to load the same binary at different base addresses in memory which was very important for OSs such as DOS that did not use virtual memory yet needed to run concurrently different programs using the same libraries. Nowadays, though, this is rarely used because once they are loaded in memory and relocated, the physical pages of the code section have a different memory content for each instance of the now-unshared binary. This triggers very large memory usage when a library such as the standard C library is loaded once by each of the many processes running on a modern OS and this is what lead instead to the now-widespread use of Position Independent Code (PIC) for shared libraries.

Indirect access by current address: PIC

```
static int g_static;
int main (int argc, char *argv[])
{
  g_static++;                          call    get_pc_in_ecx
                                       add     $0x125e,%ecx
                                       mov     0x24(%ecx),%eax
                                       add     $0x1,%eax
                                       mov     %eax,0x24(%ecx)
}
```

Another option that can be used to generate a binary relies on PIC. While the LTR code presented in previous section is position independent in the sense that it can be loaded at different base addresses and still work, it is not position independent in the sense that it needs someone to relocate it before being able to run. PIC code that does not need this relocation step can be generated easily with the `gcc -fpie -pie test.c` command.

The code generated in this case is more complex but it can be decomposed in two steps: first, lines 4 and 5 load in register %ecx the address of the data section, and then, lines 6, 7, and 8 access our variable as a constant offset from the start of this data section. The important step here is the first one since instead of relying on a relocation to be performed by the loader, the address of the data section is calculated at runtime with the `get_pc_in_ecx` function which returns in register %ecx the value of the current PC and the `_GLOBAL_OFFSET_TABLE_` constant.

The code section thus requires no runtime relocations and is mapped read-only in memory by the loader to ensure that multiple processes which map the same binary share the same underlying physical pages instead of having their own set of unshared modified pages.

4.3.2 Implementation

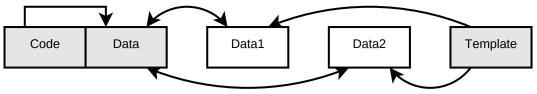
Both LTR and PIC binaries rely on a set of important common assumptions: the code and data sections of a binary are always within close distance in memory and the data section is always at a constant offset from the address of the first byte of the code section. This memory layout leads to two possible implementation options: we can load a protocol implementation binary in memory once and swap in and out a different data section whenever we execute an instance of this protocol so that the same code section sees different data section contents. This option was originally championed by COOJA. Another option involves loading the same binary multiple times in memory at different base addresses so that each code section accesses a different data section.

In this section, we thus consider two implementations:

- `DlmopenLoader`: use the `dlmopen` function to load the same binary multiple times in memory at different base addresses. See figure 4.3a.

- `CoojaLoader`: a straightforward implementation of the COOJA virtualization method. See figure 4.3b.

Code
Data1
Code
Data2
(a) Map the same binary at multiple base addresses and share the physical pages of the code section (grey)



(b) COOJA: swap data sections on event schedules

The dlmopen loader

The source code of the `dlmopen` loader is the most readable implementation of the two loaders we discuss here because it relies on the use of the `dlmopen` function which provides most of the features we need. `dlmopen` is a variant of the classic `dlopen` function that was originally implemented in the Sun Solaris OS and was later adopted by the GNU's Not Unix (GNU) C library implementation used by most Linux-based systems. Similarly to `dlopen`, `dlmopen` loads a binary in memory, executes its static constructors, and returns a `void *` handle to the newly-loaded binary. However, `dlmopen` extends the normal `dlopen` semantics by interpreting one extra *namespace* argument to decide in which *namespace* the binary should be loaded. This feature is critical because, by default, `dlopen` and `dlmopen` never load the same on-disk binary file more than once at different base addresses in one namespace. Creating multiple *namespace*s is thus the only way to ensure that the same binary is loaded more than once and that each instance of the loaded binary uses a different data section.

Each *namespace* created by `dlmopen` provides also a few other features but the most useful one is that symbols are always resolved internally within each container which makes it really easy to isolate completely a *namespace* from the other *namespace*s.

Sadly, while the implementation of the `DlmopenLoader` class is simple, the `dlmopen` implementation provided by the GNU C library does not allow the creation of more than sixteen namespaces which makes it impossible to create more than fifteen instances of a protocol implementation within ns-3 (one *namespace* is reserved as the main default *namespace*). This hardcoded constant represents the size of a statically-allocated array that can be changed only through recompilation of the system C library.

Because we felt that it would have been inappropriate to ask the ns-3 users to recompile and install a new compatible version of their system C library (it is usually much more complicated than recompiling and installing a normal library), the `DlmopenLoader` class comes also with a standalone implementation of the `dlmopen` function and the accompanying ELF dynamic loader. The implementation of this new loader is much simpler than the dynamic loader found in the GNU C library yet is both binary compatible (it could replace the system dynamic loader seamlessly) and sufficiently complete to be able to run complex applications such as the Firefox web browser on 32bit and 64-bit x86 systems running many different Linux distributions (Fedora, Debian, Gentoo, Ubuntu).

Describing the details of the implementation of this loader would be fully out of the scope of this thesis but we give below an overview of some of the most challenging aspects of this project. Interested readers will profit from [24] which gives an overview of the way an ELF loader works.

System calls: every access to system-level facilities such as `mmap`, `open`, etc. must be made through direct invocations of the associated system calls (which requires minimal CPU-specific assembly language magic). Furthermore, some of these system calls are much lower-level than the more standard UNIX versions. For example, to protect its critical sections, the loader must use the minimally-documented futex [16] system calls rather than simpler primitives such as `pthread_mutex`.

Memory management: the loader cannot rely on the `malloc` and `free` functions since they are provided by the C library and its job is precisely to load the C library. Instead, our loader wraps behind `vdl_malloc` and `vdl_free` a version of the Kingsley malloc algorithm [21] which works with blocks of memory allocated with system calls to `mmap`.

Application Binary Interface (ABI) compatibility: although traditionally the interface between the libc library and the dynamic loader was clearly well-defined, this is not the case anymore in the GNU C library implementation: over time, it has started to call more and more undocumented functions found in the loader. Furthermore, the C library directly embeds a copy of the dlopen/dlclose/dlsym family of functions: these functions naturally access all the private data structures found in the loader which makes it fully impossible to change the data structures of the loader without recompiling the C library. To work around these problems, we dynamically patch the preamble of these functions when we find them in the C library. The code inserted in these functions jumps directly inside a binary-compatible version of the same functions located within our loader.

Debugger ABI compatibility: the interface between the system debugger, the C library and the loader has traditionally been the `_r_debug` data structure which is located in the loader and allows the debugger to parse the loader link map and detect new libraries loaded by the user. While the memory layout of this data structure is relatively well established, some of the details of its semantics are much less clear: countless hours were spent to figure out what caused the debugger to crash or to report impossible values.

Workarounds: gdb but also valgrind are both very robust in general but the behavior of our loader with regard to namespaces is non-standard and this used to trigger considerable confusion in both of these tools. We learned to work around these bugs but the single biggest issue our early users reported was that their debugger behaved in very strange ways. We had thus to invest a lot of time to narrow down the root cause of these problems and submit patches to both upstream projects to fix them.

The COOJA loader

A great alternative to the complex implementation described above is the COOJA algorithm: while it is intuitively fairly CPU inefficient since every context switch requires copying entirely the data section two times (save old, restore new), it can serve as a baseline against which the `DlmopenLoader` implementation can be compared.

The original description of this algorithm found in [29] calls for a simple implementation: to ensure that the code section of a protocol implementation sees a different data section for each separate instance of the protocol, we load the relevant binaries only once in memory with `dlopen`, locate the position of their data sections and allocate a memory buffer to save a copy of the original data sections. Then, whenever we need to create an instance of these protocol implementations, we allocate a new set of buffers to save the data specific to this instance, initialize these buffers from the original data sections saved previously, and, then, copy the content of the buffers in and out of the main data section when the associated instance needs to run.

The detailed algorithm that leads to the call to `dlopen` is however a bit involved because we need to isolate the main simulation from what is going on within these protocol implementations:

1. Create the new cache directory `elf-cache/`.

2. Copy the requested binary file and its dependencies to the cache directory. For example, to load the program named `udp-perf`, we create the copy `elf-cache/udp-perf`.

3. To avoid crashes during the simulation shutdown when the protocol instances are forcibly unloaded from memory, we disable the static destructors of the binaries by removing from the on-disk copies the `DT_FINI` and `DT_FINI_ARRAYSZ` entries from the `DYNAMIC` array

4. To ensure that symbols are resolved only within this protocol instance, we modify the `DT_SONAME` and `DT_NEEDED` entries of the `DYNAMIC` array to contain unique matching identifiers that are allocated for each binary and for each dependency. For example, the `libc.so.6` `DT_NEEDED` string found in a binary is replaced by `0004.so.6` if the unique number 4 was assigned to the `libc.so.6` library and the `DT_SONAME` string found in `libc.so.6` is modified similarly so that the loader finds `0004.so.6` when searching for the dependencies of the binary.

5. To complement the above modification of the dependency names, we finally load the dependencies and the binary in inverse dependency order with the `RTLD_DEEPBIND` flag so that the dynamic loader searches the symbols first within the dependencies specified and before the global simulation scope.

4.3.3 Performance evaluation

In previous sections, we mentioned a few times the relative CPU and memory efficiency of the loader implementations presented above. In this section, we attempt to estimate quantitatively their relative efficiencies by running the same simulation with different versions of the loader. To do so, we measure first the total memory usage once all protocol implementations are loaded in memory and then the total number of packets simulated per wall clock second.

Benchmark description

We consider the linear network topology shown in figure 4.3 because is easy to setup and instantiate with a parametrized number of nodes: the left-most node sends traffic to the right-most node while every other intermediate node forwards traffic coming from its left network interface to its right network interface. To control accurately the size of the packets sent and avoid any interference from congestion control algorithms, the application traffic is encapsulated in UDP packets of a constant size.

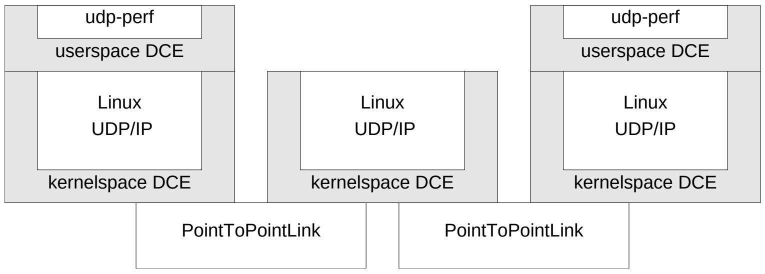


Figure 4.3: Benchmark simulation topology.

In this scenario, we use `udp-perf` to generate and receive the application traffic. `udp-perf` is a small command-line C program that uses the Linux high resolution timer [17] API and thus allows it to control accurately the transmission rate of small packets even when the target throughput is very high. `udp-perf` achieves typically much higher throughput than what a tool such as iperf [33] would report in this case because it does not use an accurate active loop that uses up CPU or an imprecise low-resolution timer API to wait between packet transmissions.

The UDP/IP network stack used in this simulation is the Linux network stack. The simulation environment that is used to provide a compatible runtime environment for the Linux network stack and the `udp-perf` application is discussed in sections 4.5 and 4.4 respectively.

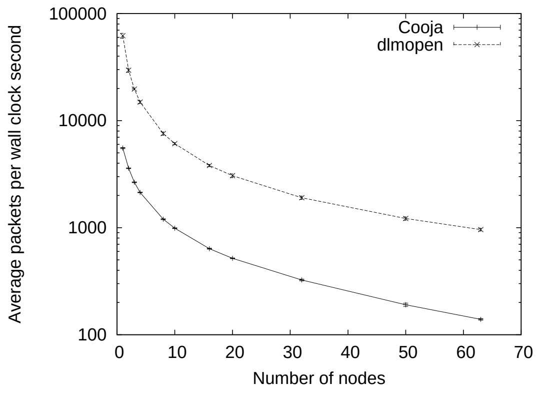
The parameters of the simulation link model were chosen arbitrarily since they have no impact on the runtime performance of the simulator: the constant delay is set to one nanosecond, the bandwidth to five megabits per second and the transmission queue is a drop tail queue with a size of one hundred packets.

Benchmark results

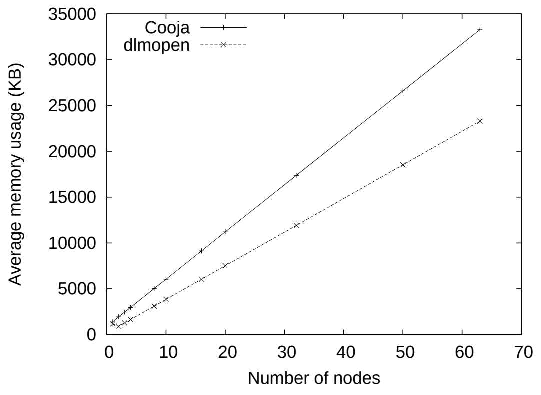
Figures 4.4a and 4.4b report the average number of packets per wall clock second and average memory usage after the protocol implementations are loaded for ten simulation runs with the 95% confidence interval. One should note that these confidence intervals are not visible, because they are too small.

Benchmark discussion

Unsurprisingly, figures 4.4b and 4.4a confirm the intuition that the overhead introduced by the COOJA algorithm translates in measurable CPU and memory inefficiencies. The extra data section template buffers which must be kept around to initialize new protocol instances for `CoojaLoader` trigger from 1.5 to 2 times higher memory usage. On the other hand, the memory copies which are needed upon each instance switch to update the data sections cost much more CPU usage, close to eight times more.



(a) Average number of packets per wall clock second



(b) Average Memory usage (KB)

Figure 4.4: The COOJA loader against the `dlmopen`-based loader.

4.3.4 Summary

In this section, we have demonstrated that a careful use of standard compiler, linker, and ELF dynamic loader features is sufficient to isolate efficiently both CPU and memory-wise multiple instances of the same protocol implementation within the same simulation process. While the resulting `DlmopenLoader` implementation is simple, it depends upon the flexible but complex implementation of the `dlmopen` function provided by VDL. Table 4.3 highlights the comparatively higher implementation complexity of the `dlmopen`-based solution implemented here as measured through the size of its C source code in KLOC.

Hopefully though, this extra implementation complexity is more than offset by the order of magnitude CPU efficiency improvement observed in real simulations over the COOJA algorithm.

	portability	C	C++	cpu efficiency	memory efficiency	code size (KLOC)
COOJA	✓	✓	✓		✓	1
`dlmopen`	✓	✓	✓	✓	✓	10

Table 4.3: Characteristics of the NSC, COOJA, and `dlmopen`-based loaders

4.4 The Linux user space emulation environment

While the dynamic loader described in previous section is the critical component of the ns-3 DCE framework which makes it possible to avoid modifying existing protocol implementations, executing one of these protocol implementations within ns-3 still requires simulation-specific replacement libraries for the C standard library, its companion the pthread library, and many other Linux-specific facilities.

Naively, one might think that implementing the 1200 functions defined in the Portable Operating System Interface [for Unix] (POSIX) [5] standard, the C library functions defined in [1], and the GNU extensions provided by the GNU C library [2] would be sufficient to create a runtime environment where most applications can execute. In practice, though, the situation is both worse and better than what the above might hint at: most applications, especially typical network protocol implementations, use a very small subset of the above functions but they also use a lot of Linux-specific extensions, if only to access the host OS routing tables and the kernel space network stack configuration facilities.

An emulation layer that is sufficiently close to a normal Linux user space runtime environment to be able to execute the BGP daemon discussed earlier in this chapter represents a large but tractable problem that is necessarily solved incrementally by identifying missing functions and adding them on an as-needed basis for the programs of interest.

The implementation presented here covers thus only partially the total set of functions available on a typical Linux distribution but it is sufficient in its current state to run unmodified the Open Shortest Path First (OSPF) implementation found in the GNU Zebra [6] routing daemon on top of the ns-3 native TCP/IP network stack. Over time, we expect to add support for more than the 240 functions already implemented but in this section, we highlight only the most challenging aspects of the ABI-compatible functions provided today.

4.4.1 Task scheduling

The thread-based programming model defined by the POSIX programming interface is fundamentally different from the event-driven model provided by a network simulator such as ns-3. For example, the POSIX `sleep` function will block, while the rest of the system keeps working, and eventually return to the caller when the right condition (at least one second elapsed) is verified later. In ns-3, on the other hand, a function that needs to wait for a specific condition would have to register a separate callback function to be invoked when the condition becomes true.

Figure 4.5 illustrates these differences through the not-so-simple example of a one second sleep. On the left-hand side of this diagram, the event-driven version of the function `Model::DoStuff` that needs to sleep after doing some work with `DoStuffStart`. To sleep, it merely needs to schedule an event for the end of the sleep and return to the simulator scheduler which will eventually execute the `DoStuffEnd` function at time 1.0.

```
void Model::DoStuff (void)
{
  DoStuffStart ();
  Simulator::Schedule (Seconds (1.0), &Model::DoStuffEnd, this);
}
```

On the right-hand side of this diagram, the thread-based version of the function `Model::DoStuff` does not need to use a secondary continuation function to complete its work with `DoStuffEnd` since `Sleep` blocks.

```
void Model::DoStuff (void)
{
  DoStuffStart ();
  TaskManager::Current ()->Sleep (Seconds (1.0));
  DoStuffEnd ();
}
```

Under the hood, though, `Sleep` is implemented with a number of secondary functions and events. When the simulation starts, instead of directly executing the `DoStuff` function from the main simulation thread, the `Model` class creates a task and associates with this task the `DoStuff` function. Later on, this function is scheduled to from an event of the main thread, starts the execution of `DoStuffStart`, and eventually enters `Sleep`. `Sleep` first schedules an event for the end of the sleep so that the `EndSleep` function is called

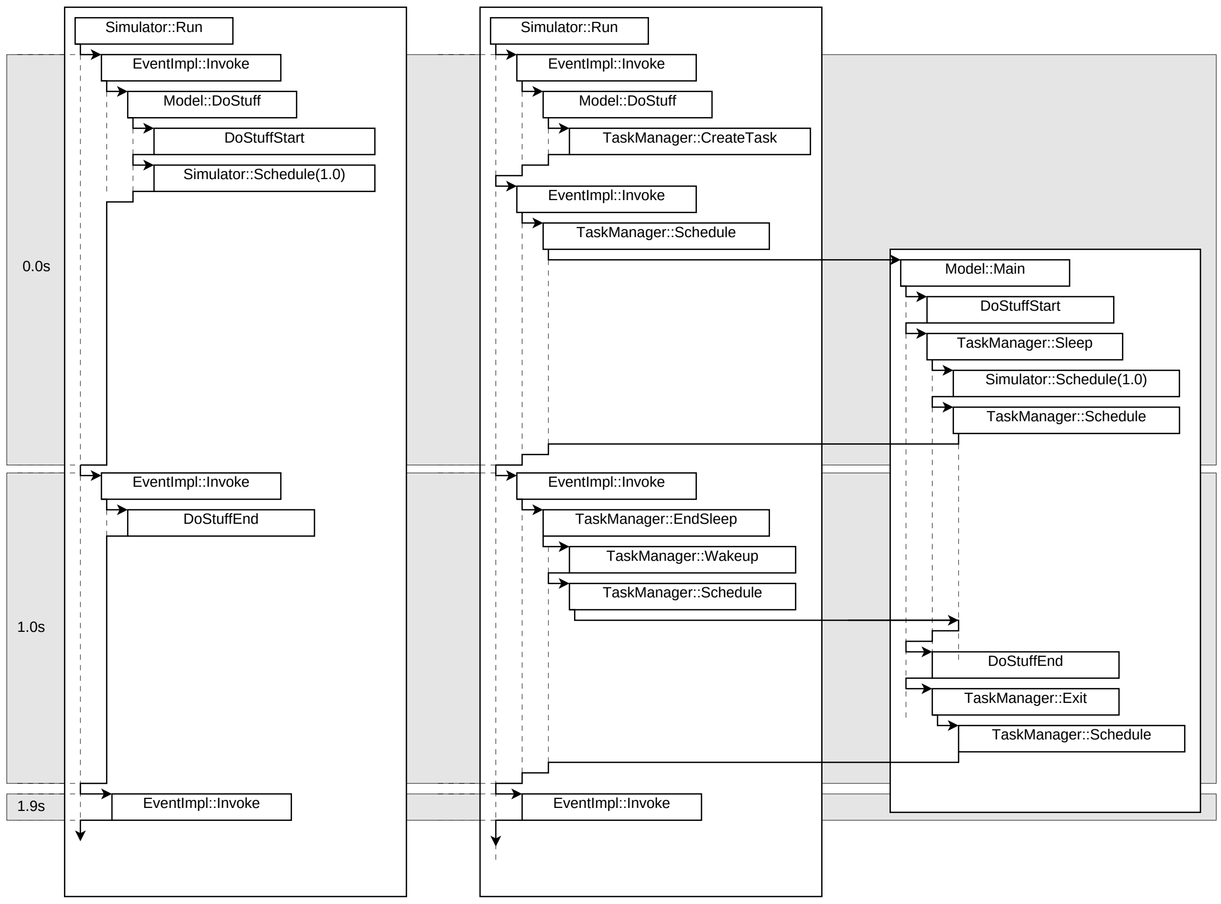


Figure 4.5: On the left, an event-driven one second sleep. On the right, a thread-based one second sleep.

when the event is executed later at time 1.0. Then, `Sleep` switches back to the main simulation thread since it has no work left to do so that the main thread can continue executing events. At some point, when the main thread reaches the event for time 1.0, it executes the `EndSleep` function which wakes up the secondary thread and then switches to the now-awake thread do restart the execution of the `DoStuff` function and allow it to finally call `DoStuffEnd` first, and then, `Exit`. The latter is eventually responsible for putting this thread in the `DEAD` state and switching one last time away from it back to the main simulator thread.

This cooperative multitasking functionality is implemented by the `TaskManager` class and uses either system-level threads or user space *fibers* [7] to manage the set of stacks used by each application. To facilitate debugging and developing new applications, we use by default the system-level thread implementation but it is easy to switch at runtime to the much faster *fiber*-based implementation.

4.4.2 Resource management

The virtualization technique we use to execute multiple applications within a single process inherently provides less isolation than more traditional platform-level virtualization solutions which can fully encapsulate an entire Operating System and protect the rest of the host system from any of its misbehavior. In the case of our DCE simulation platform, every application can write outside of its own memory areas and thus modify the state of other applications. This lack of protection is similar to embedded systems based on CPUs which lack a Memory Management Unit and thus have no way to protect a process from other processes through virtual memory. Although misbehaving applications are thus likely to wreak havoc within the whole simulation, the ns-3 POSIX emulation libraries try to provide as much isolation as possible between simulated processes by keeping track of every resource allocated by each application and carefully releasing them on behalf of the process when it exits.

For example, heap memory is an especially problematic case: in every modern C application, heap memory is allocated and freed through the `malloc` and `free` functions. C++ applications use `new/delete` which are implemented in terms of `malloc/free`. During execution, most applications are careful to post a matching `free/delete` for every `malloc/new` to ensure that no memory is leaked and thus be able to perform long-duration executions. However, a lot of applications rely on the OS to release all allocated resources when their process exits: they don't bother with invoking `free/delete` upon the process exit and, instead, assume that the OS will be able to reclaim every leaked memory allocated by `malloc` by merely un-mapping the heap pages of the process.

While we could assume that all the applications we run within our simulator are well-behaved and that they all ensure that every `malloc/new` has a matching `free/delete`, including when the process exits, doing so would be ignoring the reality of the applications we are interested in and would seriously impede our ability to run long simulations which repeatedly execute a set of applications.

We thus override the system `malloc` and `free` functions and replace them with a version which allocates memory from a process-specific pool: similarly to the dynamic loader presented in section 4.3, we use the classic malloc Kingsley algorithm (we picked [21] among the many alternatives because it is both trivial to implement and has decent performance). Our implementation allocates first fixed-size memory blocks with `mmap` and then uses the Kingsley malloc implementation to allocate user memory within these blocks. Upon process termination, we explicitly `unmap` these blocks to ensure that no memory is lost.

4.4.3 File descriptors

In POSIX applications, sockets, file accesses and a lot of other operations are performed using file descriptors: each file descriptor is an integer which is an index into a per-process table whose size is usually statically-bound. Each operation which manipulates files, directories, or, sockets uses a file descriptor to identify the object on which the operation should be applied. As such, replacing the system socket functions forces us to override all the system calls which can take as an argument a file descriptor. For example, `read`, `write`, as well as `open`, `close`, and `dup` all need to be overridden and re-implemented.

Since we have to run within the same system process multiple simulated processes running on multiple simulated network nodes, our implementation of the `open` and `opendir` functions uses a different file system *root* per simulated network node to ensure that each simulated process can read and write only node-specific files. For example, if we start two applications, A and B on node N0 and one application C on node N1, `open ("/etc/resolv.conf")` needs to access a different file on each node to make it possible to configure both nodes differently. To handle this properly, we have to keep track of the current working directory of each simulated process such that relative paths as used in `open ("foo")` are also node-specific. This leads us to also override all the functions which manipulate the current working directory, that is, `chdir`, `getcwd`.

The simplified pseudo-code for the open function shown below illustrates how these replacement functions are typically implemented:

```
int simu_open (const char *path) {
  if (path == "") {
    simu_errno = ENOENT;
    return -1;
  }
  // find an available file descriptor
  fd = allocate_simu_fd ();
  if (fd == -1) {
    simu_errno = EMFILE;
    return -1;
  }
  // prepend node-x prefix
  system_path = simu_to_system_path (path);
  // open the system file
  realFd = system_open (system_path);
  if (realFd == -1) {
    simu_errno = system_errno;
    return -1;
  }
  // update the file descriptor table
  current->openFiles.add (fd, realFd);
  return fd;
}
```

Finally, `select`, `poll`, and `epoll`, also need special treatment: contrary to [15], we implement them without performing active polling by assuming that all reads and writes to real files are always ready and that they never introduce any delay in simulation time. This decision is consistent with our focus on simulating network applications where I/O to persistent storage often has little to no impact on the application's behavior.

4.4.4 Time

The time-related functions provided by the standard C library are easy to re-implement: instead of returning the wall-clock time, we return the simulation time after converting it to the POSIX types such as `time_t`.

4.4.5 UNIX signals

UNIX signals are often compared to software interrupts as a way to notify a process or a thread of a specific event. When this event happens, the signal handler of the target process or thread is invoked and, if a system call was underway, it is interrupted and returns the `EINTR` error. By default, a handler is associated with each signal but `signal` and `sigaction` can be used to set arbitrary functions as each signal's handler. Since most of the sources of events are either irrelevant to a network application or are impossible to control outside of the kernel, the set of UNIX signals we can and need to manage, generate, and handle properly is very limited. Specifically, the only signal we attempt to deal with properly is `SIGALRM` which is generated whenever the timer started by `setitimer` expires.

A lot of networking applications use `setitimer` to ensure that they do not end up being blocked forever in a system call, waiting for remote data which will never arrive because a truck did cut a backbone optic fiber while digging for a sewage tunnel: when `SIGALRM` is delivered to the application, it interrupts the system call the application was blocked on which allows this application to continue its execution properly if it handles the associated reported `EINTR` error.

To make sure that the above-mentioned applications would work correctly when simulated, we provide an implementation of the signal handler management functions (`signal`, and, `sigaction`). We also reviewed all the functions which can be interrupted by a signal delivery (that is, every blocking function) to make sure that they perform signal delivery first and then return `EINTR`.

The former is fairly trivial since both functions merely need to update the list of signal handlers of the current process. The latter, however, is a long and error-prone process because we need to ensure that every replacement function we implement correctly returns `EINTR` when a call to `TaskManager::Sleep` returns if and only if a signal is pending, this signal is not masked, and the function has not already performed a short read or write.

4.4.6 Application Binary Interface compatibility

Although this was not an explicitly-stated hard requirement, the Linux user space emulation environment discussed here attempts to be ABI-compatible with the host Linux environment so that the same protocol implementation binary file can be run unmodified both on the host system and within the simulation.

While most of the problems related to binary compatibility are easily handled automatically by the compiler for us (it needs to use the same function calling and register allocation conventions, *etc*), this requirement introduces two major constraints:

1. The ns-3 implementation needs to provide API data structures whose size is exactly that of the host system.

2. The ns-3 replacement libraries must provide the same ELF symbol versions [9] as those provided by the host system C library to allow the dynamic loader to resolve function and global variable symbols correctly.

To illustrate the former constraint, we consider the case of the `pthread_mutex_lock` function and the associated `pthread_mutex_t` data structure. The ns-3 implementation of this function needs to store its mutex data in a data structure whose size is no larger than the size used by the host system `pthread_mutex_t`. To do so, ns-3 requires the host data structures to be bigger than 2 bytes (it is). It then stores in its first two bytes an integer handle used to search the list of `Mutex` structures stored in the current `Process` structure.

The latter constraint might seem impossible to fulfill at first because it might require us to implement all the versions of the same symbol provided by the GNU C standard library. However, the old versions of a symbol are here only for backward compatibility with old binaries that were not compiled for the host system. On standard Linux distributions, every library is always recompiled so that it does not reference the old symbols. We thus merely need to provide an implementation of the last version of each symbol. To do so, though, we need to find the version name of the last version of each symbol which changes from one Linux distribution to the other because they use different versions of the GNU C library. It is thus not possible to hardcode in our standard C library implementation the set of symbol version names and we have to extract that information from the host system ns-3 runs on before we can compile our simulation system libraries with a version description file appropriate for that host.

4.4.7 Summary

Others such as [18, 15, 11] have implemented replacement libraries for the host system libraries. However, none of these were ever made available for others to build upon and create a really useful robust DCE environment. In this section, we have described the implementation provided by ns-3 and we have attempted to highlight the considerable complexity involved in making this implementation truly robust and usable on a large range of systems. While this library is still incomplete from the perspective of replacing all the functionality provided by normal system libraries, its functionality coverage will increase over time and we believe that its current scope and the breadth already sets it apart from other similar attempts.

4.5 The Linux kernel space emulation environment

Although the user space emulation environment described in previous section is sufficiently faithful to be able to run any user space protocol implementation once the needed missing wrapper functions are added, many users are interested instead in integrating within the simulator a kernel space protocol implementation such as a TCP/IP stack. Integrating the Linux kernel space network stack directly within ns-3 is thus a convenient way to increase even further the usefulness and the realism of the ns-3 DCE environment.

Doing so makes it also easier to integrate user space network applications that make use of more or less obscure Linux facilities. For example, the netlink socket implementation provided by the ns-3 user space DCE environment to allow Linux routing daemons to access the IP configuration of the ns-3 native stack is not needed when we use instead the Linux network stack.

In this section, we discuss the implementation of a Linux kernel emulation layer which allows us to reuse a completely un-modified Linux kernel network stack within ns-3 and replace its native TCP/IP stack. Our implementation departs from other similar attempts [22, 19] in that it focuses on automating completely the integration to avoid any modification to the linux kernel source tree. We have been using this approach for more than two months to track the fast-moving `davem/net-next-2.6` Linux kernel source tree which contains the network stack patches intended for the upcoming kernel stable release and had minimal work to do to maintain this port. Over time, we expect that, as we increase the scope of our testing strategy, and as our implementation of the kernel programming interfaces improves, the amount of changes needed to keep up with the latest version of the kernel network stack will decrease and eventually drop to zero.

4.5.1 Implementation overview

The diagram shown in figure 4.6 gives a high-level perspective of a Linux kernel network stack running inside ns-3: applications which try to send or receive packets must call a socket function at the top of the network stack. Packets travel in the network stack until

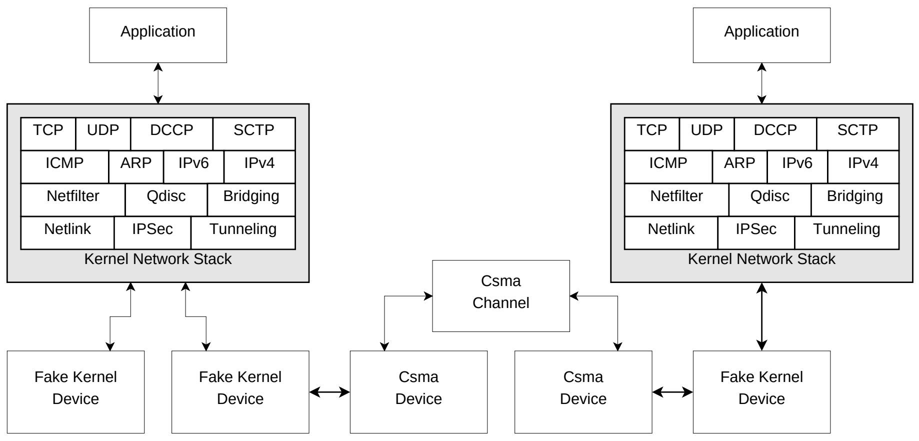


Figure 4.6: The Linux network stack running inside ns-3

they reach a *fake* kernel `net_device` which shields the real simulation `NetDevice` class from the kernel and acts as the glue between them.

Because we had no a-priori experience with the Linux kernel infrastructure other than a decent understanding of how its network stack worked, we developed the first version of these kernel wrappers incrementally by compiling and linking together all the source files from the `net` directory, and then trying to figure out which core kernel facilities were called by the network stack and how they could be re-implemented within a simulation environment. A lot of these facilities could be trivially replaced by empty stubs because they would not be actually called during a simulation or because our simulation environment is simplistic and does not need to deal with facilities such as the kernel security hooks, etc.

A few other facilities were also trivially re-implemented: this was the case for the kernel memory management primitives that were entirely replaced with simple functions that allocate memory from a pool of memory managed with [21]. This was also the case for the random number generation facilities which were forwarded to the simulator random number generator. The `printk` family of functions underwent a similar treatment. A few issues were more complex: they are thus detailed further in the following sections.

4.5.2 Build and Configuration system

Building and linking the source code present in the `net` directory seemed at first trivial. However, it proved to be very challenging because the order in which the files are linked is critical to ensure that their initialization functions are executed in the right order when we load them in memory and because the set of files to build depends on the set of configuration options used to configure the kernel.

The solution we chose at first was very simple: we created a static configuration of the kernel source tree and we wrote a parser to scan the kernel `Makefiles` to infer the linking order of the object files generated during the build. However, we eventually learned the futility of this approach when we tried to upgrade our port to the latest version of the kernel development source tree: new configuration options had appeared, some had changed, were removed which made our old static configuration invalid. The kernel `Makefile` structure had also slightly changed and broke our parser. This forced us to:

- provide instead a valid `Kconfig` file to the kernel so that the normal kernel tools such as `make menuconfig`, `make defconfig` could be used to generate a default configuration and modify it easily as needed.
- *build* the entire kernel source tree with a *fake* `Makefile` that prints in the standard output stream a set of make rules later used to actually build a core kernel library and all the associated kernel modules.

So far, our experience with this machinery has been very positive: even though we tracked constantly the kernel development tree for more than two months, we have not had to upgrade our build and configuration system.

4.5.3 Timers and Time management

The Linux kernel management of time is based on the use of the global `jiffies` variable which contains a 32bit integer that reports the number of *ticks* elapsed since the boot of the kernel. The duration of each *tick* depends on the way the kernel was configured but nowadays, it is usually configured to be one millisecond. The jiffies variable is normally increased whenever the kernel timer interrupt is triggered. This timer interrupt is also responsible for executing any expired kernel timers every ten millisecond, etc.

[22, 19] both deal with this by executing periodically a per-node event that increases the jiffies variable and relies on the Linux kernel code to deal with its internal timers. This approach suffers sadly from one major drawback: even if there are no timers scheduled to expire for the next 10 or 80 milliseconds, the simulation will keep running and executing events, just for the sake of incrementally increasing the value of this `jiffies` variable. Rather than waste time to do this, we instead configure the Linux kernel to not use periodic ticks with `CONFIG_NOHZ` and then replace entirely the kernel timer facility to schedule simulation events for each kernel timer instead of keeping track of the kernel events in a data structure separate from the main simulation event list. The resulting kernel network stack thus runs in *tickless* mode and does not waste time scheduling unnecessary events.

4.5.4 Read Copy Update and process scheduling

Read Copy Update (RCU) [26] is a synchronization algorithm used to control access to data structures shared by different processors on Shared Memory Multi Processor (SMMP) systems. It is used in the Linux kernel to efficiently manage a number of shared data structures and is based on the idea of that shared objects are never directly modified. Instead, when a part of the kernel needs to modify some RCU-protected shared data, it first creates a copy of the data, modifies safely this local copy, and then only updates atomically the pointer to the shared data to point to the new version of the data structure.

The old version of the shared data structure is kept around and is deleted only when it can be proven that no other processor is using it. This algorithm relies on the detection of *quiescent* states to infer when it is safe to delete the old versions of the data structure. Each OS where this algorithm is used defines the *quiescent* states differently but [26] summarizes the choices made by the Linux kernel and points out, among others, that context switching is a quiescent state which means that the Linux kernel scheduler notifies the RCU implementation of every context switch to allow this implementation to invoke the appropriate deletion functions for old shared data structures.

The ns-3 kernel emulation wrappers replace the Linux kernel scheduler by the simulation task scheduler described in section 4.4.1 which means that they also need to be instrumented to notify RCU of the quiescent states. Failing to do so results in hard-to-debug deadlocks triggered by the inability of RCU to detect that it can wake up RCU users.

4.5.5 Support for `/proc/sys/`

On normal Linux systems, most kernel-level parameters such as the type of TCP congestion control algorithm can be changed at runtime by writing data in a set of magic files located in `/proc/sys`: the data we write in this so-called sysctl file system goes through the kernel file system layer until it reaches a function that knows how to convert the user string into a value that can be set in a kernel variable. A simple way to allow our users to do the same in our DCE would have been to include the whole kernel file system layer together with its `proc` file system. This would have led to a large cascading increase in the size of the kernel image together with considerably more complexity to provide simulation implementations for core kernel primitives.

Instead of pulling in the entire kernel file system layer, we thus merely compile in our kernel image the sysctl support from `kernel/sysctl.c` and then, we parse directly at runtime its underlying root `ctl_table` data structures to enumerate the hierarchical tree of parameters and to invoke the right conversion function when the user attempts to set one of them.

4.5.6 Summary

Contrary to previous attempts [19, 22], the implementation of a kernel space DCE environment for the latest version of the Linux kernel we describe in this section has been successful in avoiding *any* change (except for a one-line modification of the kernel `slab.h` header) to the entire network stack and kernel tree and has shown its ability to resist the high rate of changes seen in the mainstream Linux kernel source tree.

While the current implementation temporarily lacks support for dynamic loading of kernel modules to maximize flexibility and requires users to instead reconfigure and recompile their Linux network stack to enable or disable various protocol modules, it has been able in its present form to provide a surprisingly robust and transparent replacement for the ns-3 TCP/IP stack. The memory and CPU overhead introduced by this replacement network stack will be discussed in next section but the many new features it provides over the native ns-3 stack make it mandatory in many cases: when IP fragmentation and reassembly, IPv4 and IPv6 tunneling, Stream Control Transmission Protocol (SCTP), Datagram Congestion Control Protocol (DCCP), bridging, or one of the many queuing disciplines already implemented in the Linux network stack need to be simulated, this kernel space DCE environment is the only solution available presently to ns-3 users.

4.6 Performance evaluation

When they discover the DCE environment described in this chapter, the first question most users ask is *how many nodes can I run*? In general, though, this is not the question they really want to ask: usually, they merely want to know if they can use these models with network topologies and traffic workloads that are typical to their area of expertise. Their need for simulations that span long time intervals and their tolerance for long simulation execution times vary a lot from a domain area to another and from one researcher to another. Furthermore, the answer depends on the kind of hardware platform they can use, how much memory it has, and whether its CPU is recent or not.

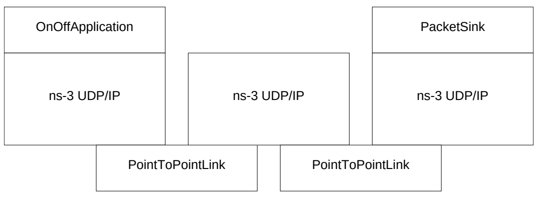
In this section, we try to answer instead two other related more specific questions:

- What is the CPU and memory overhead introduced by the use of these real-world protocol implementations compared to simpler simulation-only models?
- Once we run a simulation which contains only real-world protocol implementations, how does it compare in terms of scalability and performance with a simple testbed which uses the same protocol implementations ?

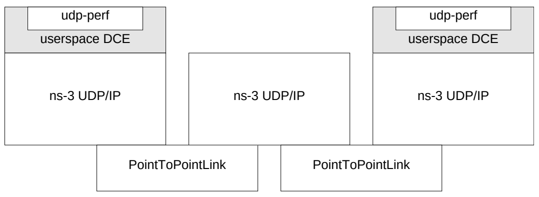
4.6.1 User space and kernel space DCE overhead

To investigate the first question, we consider again the arbitrary network topology and traffic pattern described in 4.3.3: we instantiate this scenario with different simulation models to illustrate their relative CPU and memory efficiency. Figure 4.7 gives a high-level overview of the first three experiments considered:

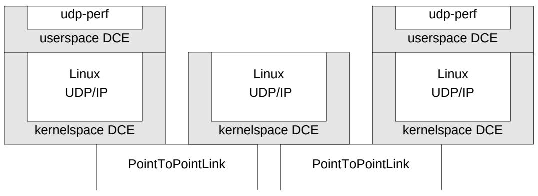
- *dce-none* (figure 4.7a): this simulation uses the native ns-3 UDP/IP stack and native `OnOffApplication` traffic generator to match the generation pattern described in section 4.3.3,
- *dce-user* (figure 4.7b): the `OnOffApplication` is replaced by `udp-perf` and the user space DCE environment.
- *dce-user+kernel* (figure 4.7c): the `OnOffApplication` is replaced by `udp-perf` and the user space DCE environment while the native ns-3 UDP/IP stack is replaced by the kernel space DCE environment for the Linux network stack.



(a) *dce-none*: ns-3 simulation with native ns-3 application and UDP/IP stack



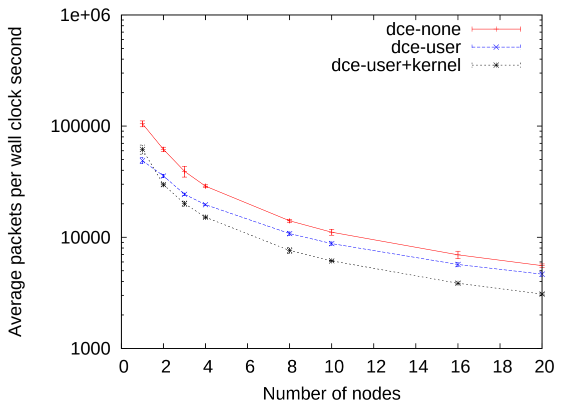
(b) *dce-user*: ns-3 simulation with `udp-perf` application and native ns-3 UDP/IP stack



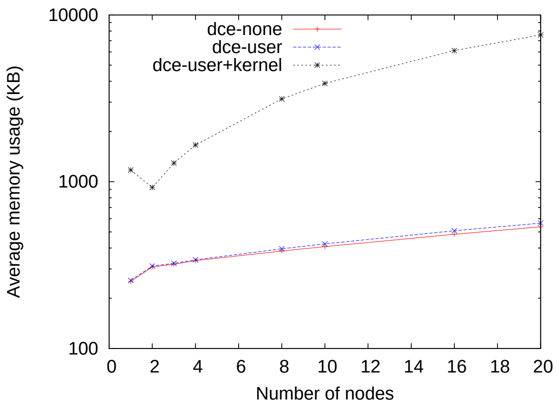
(c) *dce-user+kernel*: ns-3 simulation with `udp-perf` application and Linux kernel stack

Figure 4.7: Simulation scenario description to compare the overhead of the user space and kernel space DCE environments

Figures 4.8a and 4.8b reports the average number of packets processed in each experiment per wall clock second and average memory usage after the protocol implementations are loaded for ten simulation runs with the associated 95% confidence interval. One should note that these confidence intervals are in general invisible, because they are too small.



(a) Average number of packets (1000 bytes) per wall clock second



(b) Average Memory usage (KB)

Figure 4.8: The overhead of the DCE user space and kernel space environments

Unsurprisingly, figures 4.8a and 4.8b confirm that user space and kernel space DCE environments both contribute significant CPU and memory overhead: a simulation which integrates both processes down to two times less packets per wall clock second and consumes up to twenty times more memory.

4.6.2 Comparison with a testbed

To answer the second question, we consider a testbed based on the Linux network namespace container virtualization technology, NetNs. In next chapter, we describe a testbed based on this lightweight technology which makes it easy to create on the same physical host multiple network stacks and to interconnect these virtual network stacks by kernel-level emulated links. In this chapter, though, we conduct the simple experiment described in figure 4.9. This figure illustrates *netns*, an experiment which attempts to mirror the topology and traffic patterns setup in *dce-user+kernel*: a set of Linux NetNs network stacks are interconnected by kernel-level point to point links. The application traffic is generated again by `udp-perf`, it is processed by the virtualized network stacks, and the kernel-level packet scheduling framework enforces a constant delay and bandwidth constraint on each link.

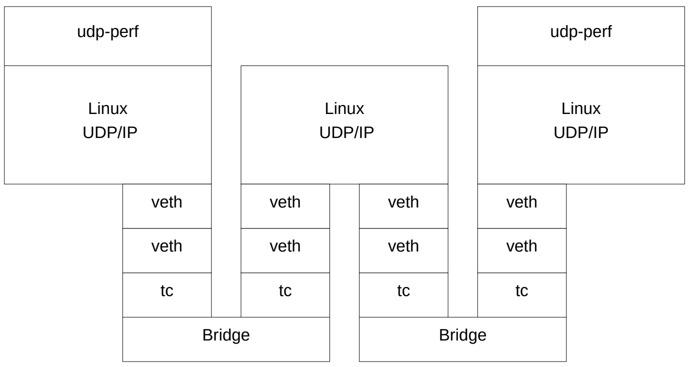


Figure 4.9: An experiment based on Linux network namespaces.

Figure 4.10 reports the number of packets received per second on the right-most node when increasing the transmission rate on the left-most node has no impact anymore on the reception rate. This value represents the maximum number of packets that the testbed system used in this experiment can process per wall clock second: it can be meaningfully compared against the number of packets simulated per wall-clock second measured in previous section and shown again in figure 4.10.

This figure suggests that while a simulation which integrates both kernel space and user space protocol implementations is slower than the same experiment running directly within the host Linux OS by a factor of two, the equivalent simulation using only native simulation models is in fact faster by a slight margin than the kernel host Linux OS.

4.7 Conclusion

In this chapter, we have outlined one of the strategies that can be used to increase the realism of a network simulator such as ns-3 by integrating protocol implementations de-

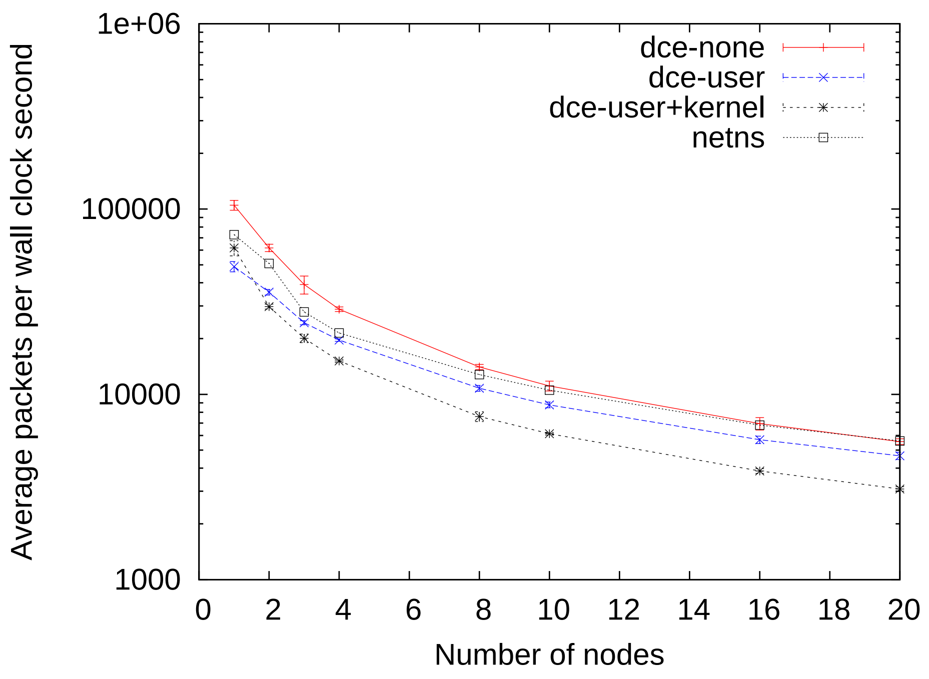


Figure 4.10: Packets per second processed by simulation vs by testbed

signed to run on real systems. Specifically, we have shown that the use of a simple ELF dynamic loader together with replacement libraries for the Linux user space environment and the kernel space runtime environment is sufficient to allow the Direct Code Execution of unmodified user space and kernel space protocol implementations. In the case of user-space protocol implementations, we have gone as far as showing that the same application binary can run both standalone on the host system to be used in a testbed or directly within the simulator. These three components combined together make it possible to easily and seamlessly switch back and forth between simulations and testbed experiments to exploit the reproducibility and debuggability of a simulation together with the high degree of realism of a testbed or a field experiment.

In some cases, though, users need to do more than switch easily between a field experiment and a simulation: they need to be able to use both at the same time within the same experiment to extend even further the scope in terms of realism of their simulation while retaining its strong reproducibility characteristics. However, trying to mix together a simulation with a more realistic testbed introduces a host of new issues which we discuss in the next chapter.

Bibliography

[1] *ISO/IEC 9899: Programming languages — C.* 1999. 87

[2] The GNU C Library. URL `http://www.gnu.org/software/libc/`. (Accessed September 5th 2010). 87

[3] System V Application Binary Interface, Intel386 Architecture Processor Supplement, fourth edition. Technical report, 1997. 77, 79

[4] Network namespace for Linux Containers. URL `http://lxc.sourceforge.net/`. (Accessed September 5th 2010). 76

[5] *IEEE 1003.1: Portable Operating System Interface (POSIX).* 2008. 87

[6] GNU Zebra. URL `http://www.zebra.org/`. (Accessed September 5th 2010). 73, 87

[7] Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur. Cooperative Task Management Without Manual Stack Management. In *USENIX Annual Technical Conference, General Track*, pages 289–302, 2002. 89

[8] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM. 76

[9] Linux Standard Base. ELF Symbol Versioning. URL `http://refspecs.freestandards.org/LSB_4.0.0/LSB-Core-generic/LSB-Core-generic/symversion.html`. (Accessed September 5th 2010). 92

[10] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association. 76

[11] Craig Bergstrom, Srinidhi Varadarajan, and Godmar Back. The Distributed Open Network Emulator: Using Relativistic Time for Distributed Scalable Simulation. In *PADS '06: Proceedings of the 20th Workshop on Principles of Advanced and Distributed Simulation*, pages 19–28, Washington, DC, USA, 2006. IEEE Computer Society. 76, 93

[12] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53 (2):66–75, 2010. 75

[13] Xenofontas A. Dimitropoulos and George F. Riley. Creating realistic BGP models. In *In Proceedings of Eleventh International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'03*, 2003. 73

[14] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors. In *LCN '04: Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, pages 455–462, Washington, DC, USA, 2004. IEEE Computer Society. 74

[15] David Ely, Stefan Savage, and David Wetherall. Alpine: a user-level infrastructure for network protocol development. In *USITS'01: Proceedings of the 3rd conference on USENIX Symposium on Internet Technologies and Systems*, pages 15–15, Berkeley, CA, USA, 2001. USENIX Association. 73, 91, 93

[16] Hubertus Franke and Rusty Russell. Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux. In *Proceedings of the Ottawa Linux Symposium (OLS'02)*, 2002. 82

[17] T. Gleixner and D. Niehaus. Hrtimers and beyond: Transforming the Linux time subsystems. In *Proceedings of the Ottawa Linux Symposium (OLS'06)*, 2006. 85

[18] X. W. Huang, Rosen Sharma, and Srinivasan Keshav. The ENTRAPID Protocol Development Environment. In *INFOCOM*, pages 1107–1115, 1999. 74, 93

[19] Sam Jansen and Anthony McGregor. Simulation with real world network stacks. In *WSC '05: Proceedings of the 37th Winter Simulation Conference*, pages 2454–2463. Winter Simulation Conference, 2005. 75, 93, 95, 96

[20] Sam Jansen and Anthony McGregor. Static virtualization of C source code. *Softw. Pract. Exper.*, 38(4):397–416, 2008. 75

[21] C. Kingsley. Description of a very fast storage allocator, Documentation of 4.2 BSD Unix release, 1983. URL xxx. (Accessed September 6th 2010). 83, 90, 94

[22] Christopher C. Knestrick. Lunar: A User-Level Stack Library for Network Emulation. Technical report, Department of Computer Science, Virginia Polytechnic Institute and State University, 2004. 76, 93, 95, 96

[23] Mathieu Lacage and Thomas R. Henderson. Yet another network simulator. In *WNS2 '06: Proceedings of the 2006 workshop on ns-2: the IP network simulator*, page 12. ACM, 2006. 73

[24] John R. Levine. *Linkers and Loaders*. Morgan Kaufmann Publishers Inc., 1999. 82

[25] Michael Matz, Jan Hubicka, Andreas Jaeger, and Mitchell Mark. System V Application Binary Interface, AMD64 Architecture Processor Supplement, version 0.99.4. Technical report, 2010. 77, 79

[26] Paul E. Mckenney. *Exploiting deferred destruction: an analysis of read-copy-update techniques in operating system kernels*. PhD thesis, 2004. Oregon Health & Science University, Supervisor-Walpole, Jonathan. 95, 96

[27] Joy Mukherjee. A compiler directed framework for parallel compositional systems. Technical report, Department of Computer Science, Virginia Polytechnic Institute and State University, 2002. 76

[28] Joy Mukherjee and Srinidhi Varadarajan. Develop once deploy anywhere achieving adaptivity with a runtime linker/loader framework. In *ARM '05: Proceedings of the 4th workshop on Reflective and adaptive middleware systems*, New York, NY, USA, 2005. ACM. 76

[29] F. Osterlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt. Cross-Level Sensor Network Simulation with COOJA. pages 641 –648, nov. 2006. 74, 83

[30] Rusty Russel and Jeremy Kerr. nfsim: Untested code is buggy code. In *Proceedings of the Ottawa Linux Symposium (OLS'05)*, 2005. 74

[31] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 1–14, Berkeley, CA, USA, 2001. USENIX Association. 76

[32] SWSoft. OpenVZ. URL `http://www.openvz.org`. (Accessed September 5th 2010). 76

[33] A. Tirumala, M. Gates, F. Qin, J. Dugan, and J. Ferguson. Iperf - The TCP/UDP bandwidth measurement tool. URL `http://iperf.sourceforge.net/`. (Accessed September 5th 2010). 85

[34] S. Y. Wang and H. T. Kung. A new methodology for easily constructing extensible and high-fidelity TCP/IP network simulators. *Comput. Netw.*, 40(2):257–278, 2002. 75

[35] Marko Zec. Implementing a Clonable Network Stack in the FreeBSD Kernel. In *Proceedings of the USENIX 2003 Annual Technical Conference*, pages 137–150, 2003. 76

[36] Xiang Zeng, Rajive Bagrodia, and Mario Gerla. GloMoSim: A Library for Parallel Simulation of Large-scale Wireless Networks. In *Proceedings of the Workshop on Parallel and Distributed Simulation*, pages 154–161, 1998. 73

Chapter 5

NEPI: an experimentation framework

In previous chapters, we have shown how simulation tools can be used to cover a large range of experimentation needs: when the native ns-3 models are not sufficiently faithful to the real world or when they do not implement certain protocol features, real protocol implementations can be embedded within the simulator, hence greatly extending its scope in terms of realism without any impact on reproducibility or debuggability.

However, in general, there are always certain characteristics of the real system which will not be taken into account by a simulator. When this happens, users turn to small-scale deployments or field experiments to obtain more realistic data about the behavior of their protocol in the wild. Sometimes, though, the monetary cost of these very realistic experiments, their higher variability and their lack of scalability contribute to make such experiments impractical.

For example, the delays introduced by packet processing overhead are rarely modeled in typical simulators: a user who intends to investigate the impact of the TCP/IP processing delays on application-level traffic over wireless links will be hard pressed to find a testbed platform with both accurate processing delay characteristics and reproducible wireless links. What is really needed in this case is an experimentation platform which provides the best of both worlds: reproducible wireless simulations together with accurate processing delays from the real world.

Another experiment which could benefit from a *mix* between simulation and the real world is a deployment of a set of real-time simulations over PlanetLab to obtain internet-style background traffic conditions for the traffic generated by the simulators running on each PlanetLab node. If the PlanetLab nodes are sufficiently powerful CPU-wise, each simulation running on PlanetLab could be used to simulate tens of nodes, hence greatly extending the scale in terms of number of nodes of the experiment without giving up on its internet-style background traffic conditions.

While these scenarios are simple to state, it is hard to make them come true: in both cases, the complexity of deploying, and configuring correctly these experiments put them out of reach of most experimenters since few of them can afford the time necessary to set them up.

In this chapter, we focus on the problem of simplifying the description and deployment of *mixed* network experiments which involve a combination of two or more of a simulation, a testbed and a field experiment. Specifically, we demonstrate the feasibility of automating entirely the configuration and deployment of the network and application level aspects of these experiments through the use of a unified experiment description scheme which relies on a box/connector object model. Because we have finite development resources it would be impossible to implement support for every experimentation tool in existence today. We thus arbitrarily chose to consider mixed experiments which involve the ns-3 simulator and the NetNs virtualization technology because of our familiarity with their inner workings.

In section 5.1, we present the limitations of the rare projects which have attempted to solve similar problems. Then, in section 5.2, we illustrate through an realistic example how painful setting up manually such a *mixed* experiment can be. Because automating the setup and deployment of these experiments requires global knowledge of their topology and configuration, section 5.3 introduces the object model defined by NEPI which can be used to describe the network-level as well as the application-level aspects of a network experiment that spans multiple experimentation tools.

5.1 Related work

While there are many network experimentation tools out in the wild, each with its own peculiarities and features, few of them really attempt to provide a solution to the problem we are interested in, that is, use a single experiment description to automate the network-level and application-level deployment and configuration across multiple independent experimentation platforms.

5.1.1 Distributed systems

Tools such as Splay [12] and Plush [5] are grounded in distributed systems research and thus are mostly focused on automating the deployment of applications, regardless of the underlying network topology.

Splay abstracts the transport layer behind a dedicated runtime library and ad hoc language. Distributed applications which are written to work within the Splay runtime can be automatically deployed on network experimentation platforms such as PlanetLab [6], ModelNet [14], Emulab [15] or arbitrary computing clusters. Similarly, Plush [5] and its followup project Gush [4] act as a wrapper on top of a set of computing resources allocated separately.

While projects such as these make it easy to deploy the same application on various experimentation platforms, they provide no control over the description and specification of a network topology and thus cannot take care of setting them up if needed.

5.1.2 MyPlC

The CoreLab [1], OneLab [3], PlanetLab [6] projects and their many offspring such as VINI [7] are all derived from the same MyPlC codebase: they control a set of internet-connected hosts on which any number of independent virtual environments denoted slivers can be created to host user-deployed applications. These testbeds are typically used to test internet services against internet-style background traffic and churn.

While it is easy to describe and automate the allocation of a set of slivers, these testbeds provide no support for application deployment and instead rely on tools such as Splay and Plush to fill in this gap. A typical experiment conducted over PlanetLab is thus usually a two-step process: first, the experimenter needs to describe and allocate a set of network resources, and then, he needs to deploy separately his application over these resources.

5.1.3 ModelNet

ModelNet testbeds allow the experimenter to describe a fine-grained network topology which is emulated by a set of Linux kernel-level software components. This emulated network topology is then used to interconnect a set of virtual machines on which user applications can be run. Rather than confront the user application with internet-style background traffic as in PlanetLab, ModelNet instead attempts to model the characteristics of its routers with no background traffic.

The user model offered by ModelNet [14] is very similar to that of the MyPlC derivatives discussed above and relies again on third party tools to automate the deployment of user applications over the ModelNet-controlled network topology.

5.1.4 OMF

cOntrol and Management Framework (OMF) [2], slowly evolved over the years from a tool used to describe, deploy, and collect the results of an experiment within the ORBIT [13] testbed to a more versatile and generic solution, able to deploy the same experiment description either on PlanetLab through its Slice Facility Architecture (SFA) [11] reservation mechanism or on an OMF-controlled testbed. An OMF experiment is described in a script which is responsible for allocating, and setting up both the network-level and the application-level elements of the experiment.

While OMF is unable to automate the deployment of a mixed experiment that involves two testbeds at the same time, it nonetheless stands out in stark contrast to the other experimentation tools discussed so far since it provides control over both the experiment network topology and the application deployment.

5.1.5 Emulab

Emulab [15] is similar to OMF in terms of scope and functionality: its experiment control tools allow the description of a network topology and the automated deployment of applications over this topology. Emulab can control access to many very different types of network resources: from virtual machines with kernel-level emulated network links, to real hardware systems. Although its experiment description and control facility is very flexible and can be used with profit to describe and automate the deployment of both the network and application aspects of an experiment over these many different experimentation facilities, Emulab requires new experimentation testbeds to be integrated within its control framework which makes it hard to reuse a testbed that is not already part of Emulab.

5.1.6 Summary

Table 5.1 attempts to provide a high-level summary of the features found in existing experimentation tools. The *application deployment* capability refers to the ability of automating the application deployment while the *network topology* capability refers to the ability of

	application deployment	network topology
Plush	✓	
Splay	✓	
MyPlC		✓
ModelNet		✓
VINI		✓
OMF	✓	✓
Emulab	✓	✓

Table 5.1: Capabilities of select experimentation tools

automating the setup of the network topology needed for the experiment. Because these two capabilities are fairly complementary, most tools provide only one and rely on other tools to provide the missing piece. However, this lack of integration means that many tasks that require a global view of every layer of an experiment cannot be easily automated and this is why OMF and Emulab both attempt to deal with both aspects of the experiment description and deployment.

However, both OMF and Emulab also require that every resource used in an experiment be under the control of their own management layer which makes it hard to use transparently new experimentation tools which are under the control of another management system. NEPI was built as an answer to that specific challenge: that of providing a uniform API to describe and deploy automatically experimentation resources located in physically distinct testbeds under the control of administratively independent entities.

5.2 Usecase

While the problem of using together a simulation with a testbed or a field test is common to every user who needs to combine the reproducibility inherent to the former with the realism provided by the latter, we consider in this section one specific example and we attempt to illustrate through this concrete example the considerable pain involved in its manual setup.

From a high-level perspective, the experiment topology we deal with here can be accurately summarized by figure 5.1: this scenario depicts a video data server located in a data center accessed by a set of video clients through a Worldwide Interoperability for Microwave Access (WIMAX) BS/SS access network. The video client and video server nodes use a virtualized network stack to execute a set of video client/server software instances while the boxes surrounded by light grey simulate the WIMAX access network. The network stack is virtualized with the Network Namespace (NetNs) lightweight virtualization technology. The WIMAX links are simulated with ns-3 to take advantage of its native native WIMAX models [8, 9], real-time scheduler and packet conversion capabilities.

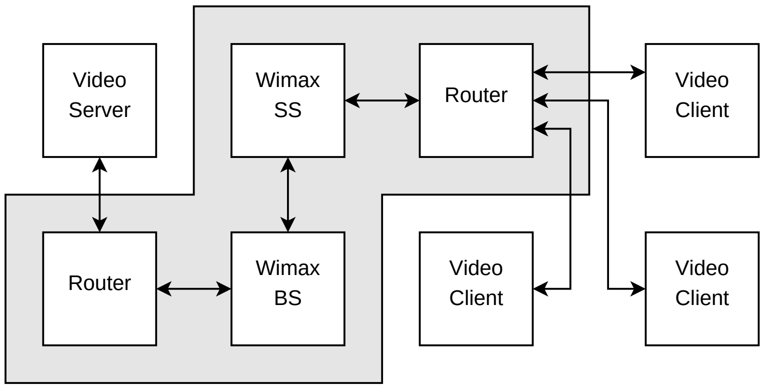


Figure 5.1: A set of virtualized network stacks interconnected by a simulated WIMAX link

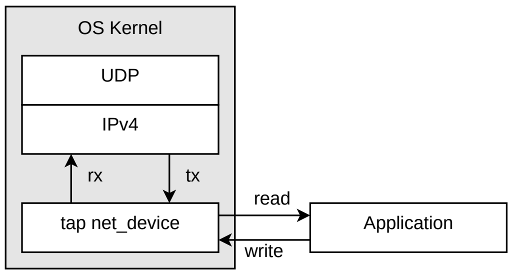
5.2.1 Tap device

The scenario described above relies heavily on the continuous exchange of packets between kernel-level network stacks and an application to simulate the layer 2 links. In chapter 3, we presented the ns-3 facility used to convert the real network packets to and from the simulation packet objects but this is not enough to setup a true communication channel between a simulator and a kernel network stack.

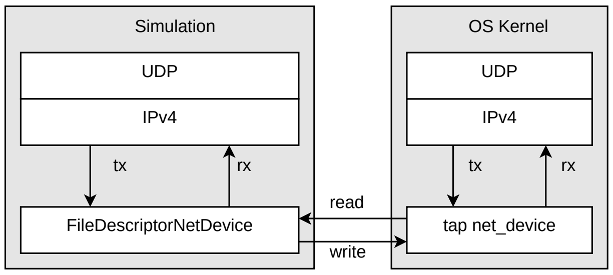
In this case, we need to extract outgoing network packets from the bottom of the kernel network stack, insert them at the bottom of the simulated network stack, and vice versa. Figure 5.2a illustrates this process when it is implemented by a tap `net_device`. When an application needs to exchange packets with the kernel, it opens the special `/dev/tun` file and performs a special `ioctl` on it to request the creation of a new tap device. The `ioctl` system call returns a new file descriptor which can then be used to manipulate the content of the transmission and reception packet queues of the tap device from the application: whenever the kernel sends a packet out through the tap device, it is enqueued at the back of its transmission queue and whenever the application reads from its file descriptor, it removes a packet from the front of the transmission queue. In the other direction, when the application writes in the file descriptor, the packet is enqueued at the back of the reception queue which is then later processed by the kernel at the next available opportunity to send it up the network stack.

5.2.2 FileDescriptorNetDevice

Once the packets are extracted from a tap device and have been received in a userspace application, they need to be injected in and extracted out of an ns-3 simulation to model the wireless links we are interested in. The `FileDescriptorNetDevice` (pictured in figure 5.2b is a `NetDevice` which was designed to work together with the tap device described above and accomplishes the mirror operation within ns-3. Once it is created, the `FileDescriptorNetDevice` waits for someone to create a tap device and hand out the associated file descriptor. Later on, any attempt by the ns-3 network stack to transmit a



(a) A tap `net_device` connected to an application



(b) A `FileDescriptorNetDevice` connected to tap `net_device`

Figure 5.2: OS Kernel to userspace application communication

packet on a `FileDescriptorNetDevice` is converted in a call to `write` on the associated file descriptor while a background thread reads packets from the file descriptor and queues them for reception on the `FileDescriptorNetDevice`.

5.2.3 Network Namespaces

To allow multiple instances of the host OS network stack to co-exist on the same machine, we rely here on the use of the Linux NetNs virtualization technology: rather than run the entire OS within a platform-level virtual machine, the Linux kernel network stack was modified to indirect every access to a global variable through an extra *namespace* pointer. Each socket, process, and kernel `net_device` hold a reference to the *namespace* they belong to so that whenever a packet is received at the bottom of the network stack through a `net_device` or at the top of the network stack through a socket, the kernel knows which instance of the network stack it should access to lookup the IP routing table, *etc.*

By default, a normal Linux system contains only one network namespace so that every process, socket, and `net_device` accesses the same network stack variables. The `unshare` system call can be used to create a new network namespace and to move the calling process to this network namespace. It is then possible to move an existing `net_device` to the new namespace, to create new processes in this namespace by asking the initial process to `fork/exec` a new one, and to create sockets within this namespace by creating them from any process which belongs to this namespace. When the last process which belongs to a specific namespace exits, the namespace is destroyed by the kernel and any `net_device` still present in the namespace is either destroyed or comes back to the main namespace.

Since it is impossible to move an existing process to an existing namespace, executing a command in a specific namespace requires the use of a command server which lives in each namespace and an ad hoc communication protocol to request the execution of commands on behalf of the message sender. Figure 5.3 illustrates the creation of a namespace and the creation of a bidirectional communication channel to control the execution of a command in that namespace.

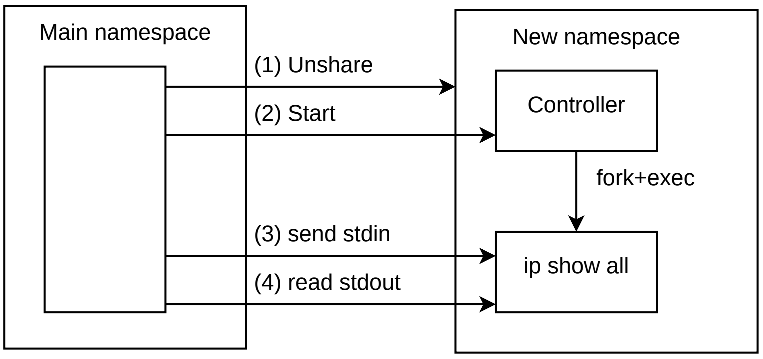


Figure 5.3: The main namespace creates a new namespace.

5.2.4 Interconnection setup

To put together a complete experiment (see figure 5.4) from the individual components described above, a user would need to perform many tasks:

- First, create a set of network namespaces for the video server and client software.
- Then, create a set of tap `net_device` and move one of them to each network namespace
- Start a real-time simulation which simulates the WIMAX link and contains one `FileDescriptorNetDevice` for each tap `net_device`
- Communicate the file descriptor of each tap `net_device` to the matching `FileDescriptorNetDevice`
- Assign IP addresses to the tap `net_device`s located in the namespaces, and the `FileDescriptorNetDevice`, WIMAX base station and subscriber station devices.
- Setup the IP forwarding tables of the network namespaces and those of the simulation IP stack

5.2.5 Summary

There is clearly nothing impossible about the set of steps presented above but, in practice, they are extremely error-prone because they require a thorough understanding of the interaction between the simulation and the network namespaces running on the host. Most users who attempt to conduct this experiment will spend countless hours to figure out how to make sure that each `FileDescriptorNetDevice` is assigned the right matching file descriptor to exchange data with the OS tap `net_device` or to track down which IP stack has been mis-configured.

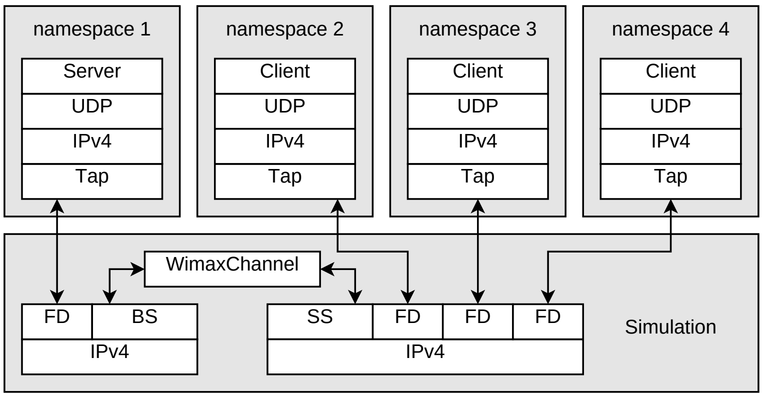


Figure 5.4: Experiment setup

The key problem in this case is that knowledge about the experiment topology is split between different scripts, written in different languages, using different programming interfaces, and this is what makes it impossible to automate the global assignment of IP addresses, the configuration of the associated IP forwarding tables, or to ensure that the tap `net_device` is interconnected with the right simulation device.

5.3 A unified object model for network experimentation

To address the problem described above, the most obvious solution is to centralize in a single script the description of the entire experiment and to use this unified description to automate the configuration steps described in 5.2.4. In this section, we present the NEPI programming interface abstraction which we have been using successfully to describe and automate the setup and deployment of both the simulation and the network namespace aspects of the experiment discussed in previous section.

5.3.1 The NEPI framework

NEPI is a python library which provides access to a set of back-ends that export the same programming interface to the user. As outlined in Figure 5.5, it is possible to interact with NEPI by writing a python script or by using the Network Experimentation Frontend (NEF) user interface.

The programming interface is accessed through an entity called a `Controller`, which runs and monitors experiments. The `Controller` can be the same process or an independent process on a possibly remote computer, allowing the user to detach and reattach to a running experiment.

Another important concept in the framework is that of a `Server`: it represents a back-end instance, such as an ns-3 process or an Emulab boss. The `Server` is the gateway that

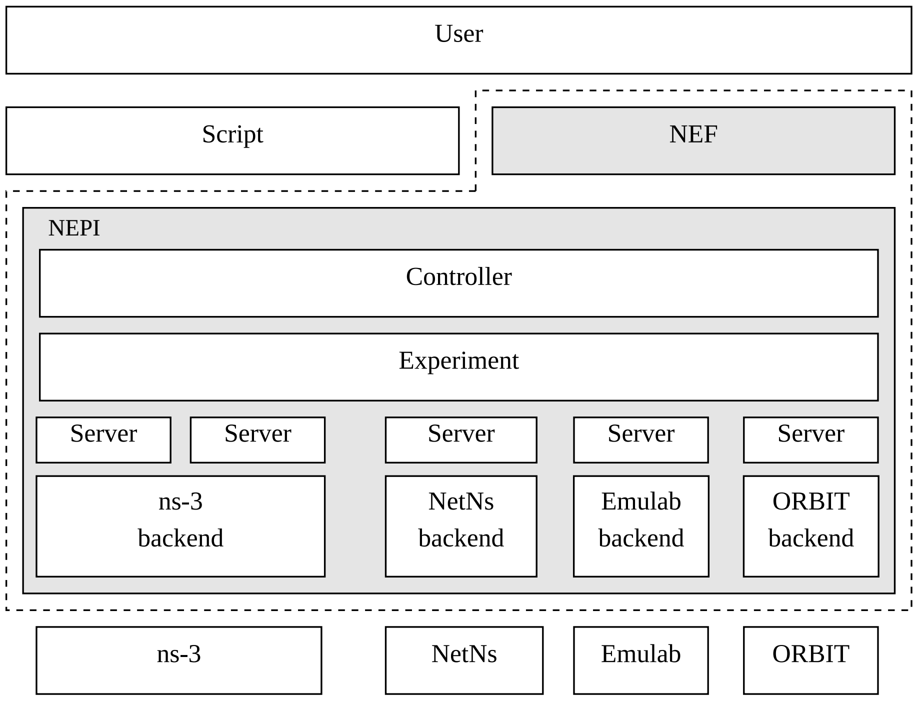


Figure 5.5: NEPI architecture

provides for the deployment of the experiment.

The `Experiment` object is responsible for keeping track of the description of an experiment and the set of servers which will execute certain subsets of the experiment. In the case of the video/client server example described in 5.2, the `Experiment` object would keep track of two server objects: the ns-3 server would represent the ns-3 process responsible for simulating the WIMAX access network while the NetNs server would represent the set of network namespaces created on a host which encapsulate the video clients and video server. To facilitate the management of multiple experiments or conduct the same experiment multiple times, the `Experiment` object can also be used to serialize and de-serialize an experiment description to/from XML files.

5.3.2 The box metaphor

The description of an experiment in Nepi is based on the box metaphor: each box represents a separate functional unit connected to other functional units/boxes through named ports. By using named ports, relationships between boxes gain semantic value that is useful both for the user and the implementation. Users familiar with IC design or board layout tools will feel immediately at ease with this model.

An experiment within a testbed can be modeled by a set of interconnected testbed-specific functional units, each of which is described by a set of specific attributes. Different testbeds provide different kinds of functional units with different attributes but the programming interface used to connect together a port within a functional unit to another port within another functional unit is the same for every back-end.

5.3.3 A concrete example

To illustrate how the box metaphor can be used to model a typical network experiment, we consider again the video client/server example described previously. Figure 5.6 contains an abstract representation of a small subset of this experiment.

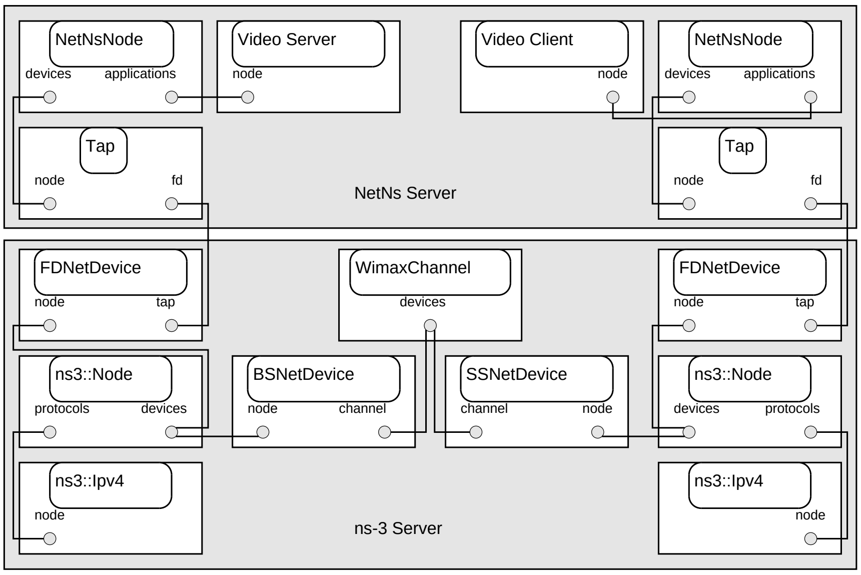


Figure 5.6: Video client/server usecase

This example highlights two important properties of the modeling methodology adopted here:

- Each backend (the ns-3 backend and the network namespace backend) exports boxes which have different semantics and different levels of abstraction: an ns-3 node does not contain an IP stack unless it is connected to the `Ipv4` box while a network namespace node encapsulates the entire TCP/IP stack of the Linux kernel.
- The semantics of each connection between a pair of ports vary depending on the two objects connected and the ports they are connected through. In some cases, a connection represents merely a function call to associate two C++ objects together (the `ns3::Node`/`ns3::Ipv4` connection for example) but in other cases, the connection represents a set of actions which need to be performed in a certain order to setup a set of unrelated objects (the `Tap`/`FileDescriptorNetDevice` connection for example).

The ns-3 and the network namespace backends shipped with NEPI take advantage of these two properties to export one NEPI box per functional unit that is accessible in the underlying tool. In the case of ns-3, every ns-3 object is exported as a NEPI box while in the case of the Linux network namespaces, since the kernel gives only control over the creation of an entire TCP/IP stack, this is the only functional unit exported as the `NetNsNode`.

5.3.4 Box composition

While the flexibility that is granted to backend implementors to chose the granularity of the functional units they can export to their users through NEPI facilitates their implementation task considerably, from the perspective of a user, it can make it fairly painful to describe a complex experiment.

In the case of the ns-3 backend, for example, describing the set of objects which are needed to run successfully a WIMAX simulation can be daunting: the illustrative diagram shown in figure 5.6 only scratched the surface of what would have been needed to describe our video client/server example. To make the diagram readable, we removed the mobility models which are needed by the WIMAX channel path loss model, we also removed the path loss model description, and various other important functional units.

In general, users thus need the ability to hide these complex low-level configuration details until they really need to change them. Each NEPI box is, of course, created with a set of default values for all its configuration attributes but what is needed here is the ability to create a set of objects in a certain state by default instead of creating one object in a default state.

The natural way to handle this issue with the box model chosen by NEPI is to extend it to support hierarchical composition of boxes so that one composite box contains an arbitrary number of internal primitive or composite boxes. NEPI supports this by allowing the user to define libraries of composite boxes. These composite boxes can be, of course, unboxed to allow access to their internal objects for configuration purposes but, in practice, we have observed that our early users rarely do this.

5.3.5 IP configuration

If we go back to the example described in 5.2, it should be clear that the box model presented here provides sufficient information to solve some but not all of the automation objectives we set out to solve. For example, the existence of the triplet `Tap`, `FileDescriptorNetDevice` and tap-device connection is sufficient to automate the creation of the kernel tap `net_device`, the ns-3 device object, and the exchange of the communication file descriptor between the tap `net_device` and the ns-3 device. However, none of the abstractions we have discussed so far provide any clue to NEPI about the IP-level topology of the experiment, hence making it problematic to automate the assignment of IP addresses to network interfaces and the configuration of the IP forwarding tables in each experiment node.

To deal with this issue, NEPI requires every backend to provide for each box it exports a way to identify its local IP topology. The topology object associated with each box is then queried at runtime before starting the user experiment to construct a global description of the experiment IP topology. Once constructed, this topology description is finally parsed to assign one IP address per network interface and to configure the associated forwarding tables. The algorithm used to assign IP addresses is currently naive and inefficient since it simply allocates consecutive IP addresses and thus requires n entries in each forwarding table (where n is the number of network interfaces in the entire experiment) but we plan to integrate in the future more efficient alternatives such as [10].

Deferring the assignment of IP addresses to this late stage of the experiment setup means, however, that it is not possible anymore to use IP addresses to specify communication endpoints. Say, if we create a `Ping` box and connect it to one of the node boxes of our experiment to specify where the `ping` program should run, we are now unable to specify the destination IP address of the associated Internet Control Message Protocol (ICMP) echo request because we do not know which IP address will be assigned to the remote node we intend to ping.

In Emulab, the *boss* server addresses this problem by dynamically creating a set of Domain Name System (DNS) entries which can be used to identify each host deployed in an experiment. In the case of NEPI, this method is hard to implement because it would require each backend (including simulation backends) to be able to deploy automatically a DNS server within each experiment. Instead, NEPI adopts a simpler approach: it associates with each entity visible within the IP topology a unique hierarchical string description and converts these string descriptions to actual IP addresses once the IP address assignment has been completed, hence allowing users to alias IP addresses with these strings.

5.4 Summary

Although NEPI is still a work in progress because it lacks support for PlanetLab, ModelNet, or EmuLab backends and because it has, so far, received little testing outside of the tighly-knit group of researchers within the Planete team, it has been successful at solving the problem we set out to address. Specifically, we believe that it has demonstrated the feasibility of automating entirely a number of tasks which usually need to be conducted manually, hence greatly facilitating the description, and deployment of network experiments based on a mix of simulation, field experiments and testbeds.

Bibliography

[1] CoreLab. URL `http://www.corelab.jp/`. (Accessed September 5th 2010). 107

[2] cOntrol, Management and Measurement Framework (OMF). URL `http://omf.mytestbed.net/`. (Accessed September 5th 2010). 108

[3] OneLab. URL `http://www.onelab.eu/`. (Accessed September 5th 2010). 107

[4] Jeannie Albrecht and Danny Yuxing Huang. Managing Distributed Applications using Gush. *Proceedings of the Sixth International Conference on Testbeds and Research*

Infrastructures for the Development of Networks and Communities, Testbeds Practices Session (TridentCom), May 2010. 107

[5] Jeannie Albrecht, Ryan Braud, Darren Dao, Nikolay Topilski, Christopher Tuttle, Alex C. Snoeren, and Amin Vahdat. Remote control: distributed application configuration, management, and visualization with plush. In *LISA'07: Proceedings of the 21st conference on Large Installation System Administration Conference*, pages 1–19, Berkeley, CA, USA, 2007. USENIX Association. 107

[6] Andy Bavier, Mic Bowman, Brent Chun, David Culler, Scott Karlin, Steve Muir, Larry Peterson, Timothy Roscoe, Tammo Spalink, and Mike Wawrzoniak. Operating system support for planetary-scale network services. In *NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, pages 19–19, Berkeley, CA, USA, 2004. USENIX Association. 107

[7] Andy Bavier, Nick Feamster, Mark Huang, Larry Peterson, and Jennifer Rexford. In VINI veritas: realistic and controlled network experimentation. In *SIGCOMM '06: Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 3–14, New York, NY, USA, 2006. ACM. 107

[8] Jahanzeb Farooq and Thierry Turletti. An IEEE 802.16 WiMAX module for the NS-3 simulator. In *Proceedings of the SIMUTools Conference.* ICST, 2009. 109

[9] Mohamed Amine Ismail, Giuseppe Piro, Luigi Alfredo Grieco, and Thierry Turletti. An Improved IEEE 802.16 WiMAX Module for the ns-3 Simulator. In *Proceedings of SIMUTools Conference.* ICST, 2010. 109

[10] John Byers Jay Lepreau Jonathon Duerig, Robert Ricci. Automatic IP Address Assignment on Network Topologies. Technical report, University of Utah, School of Computing, 2006. 117

[11] Scott Baker Tony Mack Reid Moran Larry Peterson, Soner Sevinc and Faiyaz Ahmed. Planetlab Implementation of the Slice-Based Facility Architecture. Technical report, Princeton University, 2009. 108

[12] Lorenzo Leonini, Étienne Rivière, and Pascal Felber. SPLAY: distributed systems evaluation made simple (or how to turn ideas into live systems in a breeze). In *NSDI'09: Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, pages 185–198, Berkeley, CA, USA, 2009. USENIX Association. 107

[13] D. Raychaudhuri, M. Ott, and I. Secker. ORBIT radio grid testbed for evaluation of next-generation wireless network protocols. pages 308 – 309, feb. 2005. 108

[14] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostić, Jeff Chase, and David Becker. Scalability and accuracy in a large-scale network emulator. *SIGOPS Operating Systems Review*, 36(SI):271–284, 2002. 107, 108

[15] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. *SIGOPS Operating Systems Review*, 36(SI):255–270, 2002. 107, 108

Chapter 6

Conclusion

Over the past few chapters, we have invested a lot of effort to describe our work and contributions as a sequence of logical steps in an attempt to facilitate your reading experience.

However, the daily life of a researcher is usually closer to the mess of a spaghetti plate than a beautiful mathematical demonstration. In this last chapter, before closing this thesis with some of the open questions we would like to pursue further in the future, we would like to take the opportunity to go back in time and recollect nostalgically the path that led here.

6.1 Recollections

A few years ago, when we started along the path of a Ph.D thesis, we did not really intend to work on tools for network research experimentation. Instead, we were initially focused on the problem of extracting useful topological information out of a live network without impacting its behavior and we were quickly faced with the problem of figuring out how we could assess the correct behavior and efficiency of any topology measurement protocol we could come up with.

Within a couple of months, we got sidetracked by this secondary issue and started the development of the prototype network simulator Yans with an eye towards implementing a DCE framework that could be used to host our new measurement protocols before deployment in the real world.

In hindsight, it is pretty clear that we could have pursued our initial research objective by implementing our protocols not once but as many times as needed to test them in different environments but the alluring potential of being able to reuse the same implementation within simulations and testbeds steered us away long enough to meet the ns-3 team and give up definitively any pretense of doing real network research.

Instead, we then pursued two major objectives: first, we invested considerable development time and resources in the development of the ns-3 simulator itself. We ported our event scheduler, IP stack and wireless models from Yans, implemented the ns-3 object model, attribute system, and tracing infrastructure. Finally, we rewrote entirely the packet class found in Yans to support transparent pretty-printing, flexible tagging, and more efficient operation memory and CPU-wise.

In parallel to these low-level software infrastructure facilities, we started the development of what became later the ns-3 DCE framework. It was based originally on a process scheduler written for Yans and on a simple loader written during an eight-month stay at the University of Washington within the Fundamentals of Networking Laboratory. It then slowly but surely evolved to the more efficient `dlmopen` loader. It is only much later that it gained its kernel space emulation support and its out of the box support for a variety of user space routing daemons as well as the entire Linux kernel network stack.

About two years ago, when early versions of the ns-3 DCE environment became functional, we started extensive testing and this is when we discovered the excruciating pain involved in setting up mixed experiments that use a simulator and a testbed. This discovery led to the design of NEPI which was implemented by Martin Ferrari and Alina Quereilhac.

6.2 Impact

While the road that led to this last chapter was bumpy and narrow, we believe that we were still able to improve in a useful way the daily workflow of network research experimenters through the addition of two new tools to their toolbox.

First and foremost, we believe that our contributions to the core infrastructure of ns-3 were partly instrumental to its success and its subsequent broad and large impact on the network community.

Second, we expect that, as it continues to mature, and when it is distributed with ns-3 proper, the unusually large scope of the DCE framework will increase further the impact of ns-3 on its users and the practice of network experimentation in research.

Finally, although it is harder to quantify the potential usefulness of NEPI on the network research community at large, we have been using it internally with success and we believe that the new range of experiments it enables could also have a wide impact beyond our small research group.

6.3 Future work

One of the great side-effects of our long walk on the windy path we followed until now is that we also discovered many potentially fun, interesting, and useful research directions which we decided to leave behind to be able to finish this thesis. Now that we are done, it is our hope that we will be able to investigate some of these problems further.

6.3.1 Distributed simulations

Although many time synchronization algorithms have been published and have been used to implement distributed simulation runtimes for research purposes, real implementations are usually very primitive and merely use some of the oldest conservative synchronization algorithms.

However, whether these simulators provide conservative or optimistic algorithms, few users appear to ever take advantage of these facilities when they are available because doing so requires a complex manual static partitioning of the user network topology to the available computing resources. The considerable technical expertise needed to create such a static partition and obtain measurable speedups makes these tools anecdotic.

During a 2 month internship, Guillaume Seguin demonstrated that it is possible to take advantage of the properties of SMMP systems to implement a thread-based conservative time synchronization algorithm that is entirely transparent to the user and that exhibits potentially interesting performance. His internship demonstrated the difficulty of managing thread-safety in the core simulation engine and of making this thread-safe support transparent to model developers and users but it identified a number of solutions with a prototype implementation. Since then, we have slowly integrated some of these changes back in the main ns-3 distribution and we believe that it should now be possible to investigate further the problem of adding transparent support for multi-threaded simulation to ns-3 and thus make these time synchronization algorithms more widely used.

Another potentially useful area of research involves extending this early work to support more advanced optimistic time synchronization algorithms in ns-3 with process-based snapshots using `fork`. While there is a lot of literature on these algorithms, and they appear to have been studied a lot with synthetic simulation scenarios, we are not aware of any practical implementation which can be used nowadays in production network simulators and we believe that an implementation of a few classic optimistic time synchronization algorithms would be invaluable to start a systematic analysis of the performance of these algorithms on non-synthetic simulation scenarios.

6.3.2 Aspect-based tracing

In line with these concerns about the lack of usability of existing distributed simulators, we noticed early on that while collecting traces about the behavior of a simulation is one of the most important aspects of setting up a simulation, it conflicts directly with the idea of integrating *un-modified* existing protocol implementations within the simulator through a DCE framework.

In practice, we have observed that the users of our DCE framework quickly learn the value of so-called `printf` tracing by inserting calls to this function or its C++ equivalent in strategic locations of the source code and recompiling everything whenever they change.

A relatively widespread solution to this problem adopted notably by Java programmers is to use Aspect Oriented Programming (AOP) to dynamically instrument a function of interest and be notified whenever it is called or returns. This kind of technology is however extremely complex to implement in C and C++ programs where there is no bytecode and no rules about memory consistency or multi-threading. For example, Paradyn [2] has to use multiple levels of trampolines, save and restore cpu registers by hand, and emulate the execution of some assembly instructions to allow arbitrary probes to be inserted at any time independently of the number of running threads. Sadly, neither Paradyn nor any of its competitors such as Dtrace [1] or Systemtap [3] are widely available today. Furthermore, they all use ad hoc languages to describe the location of all probes and to specify the processing to perform within these probes: when they are available, they are thus fairly cumbersome to use since all probe specifications and probe processing must be separate from the simulation itself.

Hopefully, in the case of running a simulation within ns-3, it is possible to restrict ourselves to a much simpler problem: we can assume that no probes will be inserted once the simulation starts and that threads will be synchronized safely by other means (if there are any). These simplifying assumptions make it possible to use a much simpler probe insertion mechanism based on traps which was implemented within our DCE framework and which allow us to insert a probe at arbitrary memory locations. However, to fully realize the potential of this facility, we also need to locate the right memory locations dynamically based on a user description of the probe location. For example, if the user asks for a probe at line 123 of file foo.cc, we need to retrieve in memory the position of the first assembly instruction which was generated by the compiler for that line in this file. Fortunately, the information necessary to calculate this mapping is already generated automatically by the compiler and stored in the debugging information entries of the executables and libraries it creates.

A promising approach to provide aspect-oriented programming in ns-3, and thus allow the tracing of unmodified protocol implementations, would take advantage of these debugging information entries in conjunction with the probe technique already implemented and we are hopeful that a practical solution to this problem is within reach.

Bibliography

[1] Bryan M. Cantrill, Michael W. Shapiro, Adam H. Leventhal, and Sun Microsystems. Dynamic instrumentation of production systems. In *Proceedings of the USENIX 2004 Annual Technical Conference*, pages 15–28, 2004. 124

[2] Jeffrey Hollingsworth, Barton P. Miller, and Jon Cargille. Dynamic Program Instrumentation for Scalable Performance Tools. In *Proceedings of the IEEE Scalable High Performance Computing Conference*, pages 841–850, 1994. 124

[3] Vara Prasad, William Cohen, Frank Ch. Eigler, Martin Hunt, Jim Keniston, and Brad Chen. Locating system problems using dynamic instrumentation. *Proceedings of the Ottawa Linux Symposium*, 2005. 124

Glossary

Nef Network Experimentation Frontend. 113

Nepi Network Experimentation Programming Interface. i, iii, v, 106, 109, 113–117, 123

`net_device` In the Linux kernel, the `net_device` data structure represents a network card: packets can be queued for transmission by the kernel and for reception by the hardware. 110–113, 116

ABI Application Binary Interface: an ABI describes the low-level interface between an application (or any type of) program and the operating system or another application. ABIs cover details such as data type, size, and alignment; the calling convention, which controls how functions' arguments are passed and return values retrieved; the system call numbers and how an application should make system calls to the operating system. 77, 79, 83, 87, 92

AOP Aspect Oriented Programming. 33

API Application Programming Interface: an abstraction that describes an interface that is exported by a program or library to be used by another program or library. 8, 16, 31, 53, 56, 64, 66, 73, 92, 109

ARP The Address Resolution Protocol (ARP) is used by the Internet Protocol version 4 (IPv4) to convert IPv4 addresses to MAC-level addresses. 36, 38

BGP Border Gateway Protocol. 70–73, 87

BS Base Station: a WIMAX term to designate the central static hub with which all communication is performed. 109

BSD Berkeley Software Distribution: a UNIX Operating System which is a derivative of the original AT&T UNIX Operating System. 73, 74

COW Copy-On-Write. 57, 59, 60, 63

CPU Central Processing Unit: the CPU is typically implemented in a single chip that is responsible for executing the instructions of a computer program. iii, v, 8, 17, 20, 30, 44, 45, 51, 57, 63, 72, 74, 76, 78, 79, 82–86, 89, 96–98, 106, 122

DCCP The Datagram Congestion Control Protocol is a message-oriented transport-layer protocol. 97

DCE Direct Code Execution: refers to the ability of a simulator to execute directly within itself in simulated time an program that was not originally developed to be run in a simulator. 6, 7, 71, 72, 76, 86, 89, 93, 96–98, 122–124

DML Domain Modeling Language. 31

DNS The Domain Name System. 117

ELF Executable and Linkable Format: a common binary object file format used on many UNIX systems. Defined originally in the System V ABI, later in the Tool Interface Standard with processor-specific supplements. 8, 72, 78, 79, 82, 86, 92, 100

FreeBSD A UNIX Operating System which is a derivative of the BSD UNIX Operating System. 73, 76

GloMoSim GloMoSim is a scalable simulation environment for wireless and wired network systems. It was designed using the parallel discrete-event simulation capability provided by Parsec. GloMoSim is now superseded by the commercial QualNet network simulator. 17, 18, 31, 51–53, 66, 73

GNU GNU's Not Unix: a recursive acronym that identifies the GNU project whose goal is to build a UNIX-like Operating System. 8, 82, 87, 92

GTNetS The Georgia Tech Network Simulator is a full-featured network simulation environment that allows researchers in computer networks to study the behavior of moderate to large scale networks, under a variety of conditions. 16, 17, 24, 27, 28, 31, 33, 36, 38, 47–50, 52, 53, 56, 64–66, 73

ICMP The Internet Control Message Protocol. 117

IP The Internet Protocol is the main inter-networking protocol deployed today that makes it possible to interconnect packet-switched networks together. 2, 12, 31, 36, 38, 44, 47–49, 51, 53–55, 60, 62, 64, 65, 70–73, 75, 76, 85, 87, 93, 96, 97, 106, 111–113, 115–117, 122

ISP Internet Service Provider. 70

JNI Java Native Interface function calls allow Java code running within a Java Virtual Machine to make direct calls to native C functions. 12, 74

JVM Java Virtual Machine. 33

KLOC One thousand Lines of source Code. 52, 66, 86

Linux Linux is a UNIX-compatible Operating System. 7, 8, 36, 51, 70–72, 74, 76, 79, 82, 85, 87, 92–97, 99, 100, 108, 111, 115, 116, 122

LTR Load-Time Relocation: a load-time relocation is an entry in the relocation table that is used by the dynamic loader to modify the image of the binary file once it is mapped in memory. An executable can be said to be Load-Time Relocatable if its relocation table is non-empty. 80, 81

LXC LinuX Containers is a virtualization technology integrated in recent versions of the Linux kernel that makes it possible to create multiple separate instances of the same host kernel within that host kernel. 72

MAC Medium Access Control: a sub-layer of the second layer of the seven-layer OSI model. 4, 5, 36, 44, 53, 54

MPLS Multi Protocol Label Switching. 54

NetNs A Network Namespace is the network component of a LinuX Container: it is used to virtualize access to the global variables of the Linux network stack so that multiple network stacks can be instantiated within the same Linux host. 72, 99, 106, 109, 111, 114

NSC Network Simulation Craddle. 36

OMF cOntrol and Management Framework. 108, 109

OMNeT++ The Objective Modular Network Testbed ++ is a C++ discrete event simulation environment. Its primary application area is the simulation of communication networks, but because of its generic and flexible architecture, is successfully used in other areas like the simulation of complex IT systems, queuing networks or hardware architectures as well. Its name is based on the OMNeT simulator, written in Object Pascal. 13, 16–19, 24, 27, 30, 31, 34, 35, 37, 38, 48–50, 52, 53, 56, 64–66

OS Operating System: the low-level software layer that is responsible for coordinating and protecting access to the hardware and software resources available to user programs. 17, 71, 74, 76, 79, 80, 82, 87, 89, 96, 99, 111, 112

OSPF Open Shortest Path First is an IP interior routing protocol, that operates within a single autonomous system (AS). 87

PC The Program Counter is the CPU register that contains the address of the currently-executing instruction. 77, 81

PHY Physical layer: the first and lowest layer in the seven-layer OSI model. 36

PIC Code is said to be Position Independent when its assembly instructions do not refer directly to absolute memory locations whose address can change at runtime. In short, the address at which Position Independent Code executes does not influence its behavior. 8, 80, 81

POSIX Portable Operating System Interface [for Unix]: a family of IEEE standards that specify (among others) the application programming of a UNIX OS. 87, 89–91

QoS Quality of Service. 62

RCU Read Copy Update is a synchronization algorithm used to control access to data structures that are shared by multiple readers and writers. 95, 96

SCTP The Stream Control Transmission Protocol is a message-oriented transport-layer protocol that ensures reliable in-sequence delivery of messages with congestion control. 97

SFA Slice Facility Architecture. 108

SMMP Shared Memory Multi Processor. 95, 123

SS Subscriber Station: a WIMAX term to designate potentially-mobile WIMAX nodes which are connected with a Base Station. 109

SSF Scalable Simulation Framework. 31

Tcl Tool Command Language. 13, 31, 47

TCP The Transmission Control Protocol guarantees the reliable ordered delivery of a stream of bytes over IP. 2, 3, 5, 12, 28, 34, 36, 45, 55, 59, 60, 62, 64, 70–73, 75, 87, 93, 96, 106, 115, 116

UDP User Datagram Protocol. 36, 38, 49, 51, 53, 64, 65, 84, 85, 97

UNIX UNIX is a multi-tasking multi-user Operating System. Nowadays, the term UNIX often refers instead to a family of OSes and to the programming interface that they offer. 70, 71, 76, 79, 82, 91

VDL The Virtualizing Dynamic Loader provides facilities to load the same binary more than once in memory with different base addresses so that each version of the binary loaded in memory accesses private copies of its global and static variables. 72, 86

WIMAX Worldwide Interoperability for Microwave Access. 109, 112, 114, 116

Yans Yet Another Network Simulator was originally developed at the INRIA as a prototype to demonstrate the feasibility of a number of ideas such as the use of a 64-bit integer for time, simulation packets that contain real network bytes, *etc.*. 7, 13, 28, 31, 36, 49–53, 56, 63–66, 73, 122

Zebra GNU Zebra is free software that manages TCP/IP based routing protocols such as BGP-4 as well as RIPv1, RIPv2 and OSPFv2. 87