FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Improving ns-3 Emulation Performance for Fast Prototyping of Network Protocols

Tiago Bluemel Cardoso

U.PORTO

FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Prof. Manuel Alberto Pereira Ricardo (PhD)

Second Supervisor: Helder Martins Fontes (MSc)

June 26, 2015

Improving ns-3 Emulation Performance for Fast Prototyping of Network Protocols

Tiago Bluemel Cardoso

Mestrado Integrado em Engenharia Informática e Computação

June 26, 2015

Abstract

The development of new protocols for communication systems usually starts in a network simulator, such as ns-3, where the variables that influence the network scenario can be easily controlled to create specific and reproducible test conditions. The results of such simulations are then analyzed and used to tweak and improve the protocol. After this phase, the protocol must also be tested in a real environment to obtain more accurate and credible results. To do this, the simulation code must be ported to a real system. This process of porting code from the simulation to the implementation of the protocol leads to an increase in development time and in the chance of introducing errors.

Fast prototyping is a protocol development process that attempts to solve this problem by reusing ns-3 simulation code for the implementation. This is possible because ns-3 provides emulation capabilities that allow nodes inside the simulator to communicate with those outside through an emulated network device. The problem with this approach is that emulation introduces overhead to packet processing which degrades the node's performance limiting the amount of network traffic that can be processed.

We propose an approach to reduce the performance problem associated with fast prototyping that consists in migrating the data plane operations processing to outside of ns-3. In a network node, there are two planes of operation: control and data. The control plane is responsible for discovering and maintaining network routes and ensuring connectivity. The data plane uses the routing information generated by the control plane to forward network packets. In a well designed network, most of the traffic corresponds to data. By moving the data plane operations outside of ns-3 the overhead associated with this kind of traffic is greatly reduced.

To validate our proposed solution, we extended the Wireless Metropolitan Routing Protocol (WMRP) and Optimized Link State Routing (OLSR) protocols to use the developed architecture, tested their performance in real environments and verified the amount of code reuse between the simulator and the real system.

Resumo

O desenvolvimento de novos protocolos de comunicação começa tipicamente com recurso a um simulador de redes, como o ns-3, onde as variáveis que influenciam o cenário da rede são facilmente controladas para criar condições de teste específicas e reproduzíveis. Os resultados da simulação são então analisados e usados para ajustar e melhorar o protocolo. Posteriormente a esta fase, o protocolo deve também ser testado em ambiente real de forma a serem obtidos resultados mais precisos e credíveis. Para isso, o código anteriormente desenvolvido para simulação tem que ser reimplementado num sistema real. Este processo conduz a um aumento do tempo de desenvolvimento e da hipótese de introdução de erros na implementação.

A prototipagem rápida é um processo de desenvolvimento de protocolos que tenta resolver este problema através da reutilização do código de simulação ns-3 em sistemas reais. Esta reutilização é possível porque o ns-3 disponibilizada funcionalidades de emulação que permitem que os nós simulados comuniquem com o exterior da simulação. No entanto, a emulação degrada o desempenho dos nós o que limita a quantidade de tráfego de rede que pode ser processada.

Neste trabalho, propomos uma abordagem para reduzir os problemas de desempenho associados à prototipagem rápida que consiste em migrar as operações do plano de dados para fora do ns-3. Há dois planos de operações num nó de rede: controlo e dados. O plano de controlo é responsável por descobrir e manter as rotas de rede e assegurar a conetividade. O plano de dados usa a informação das rotas gerada pelo plano de controlo para encaminhar os pacotes de rede. Numa rede típica, a maioria do tráfego corresponde a dados. Mover o plano de dados para fora do ns-3 pode, então, reduzir o custo associado ao processamento deste tipo de tráfego.

De forma a validar a solução proposta, estendemos os protocolos Wireless Metropolitan Routing Protocol (WMRP) e Optimized Link State Routing (OLSR) para usarem a arquitetura desenvolvida, testámos o seu desempenho em ambientes reais e verificámos a quantidade de código que foi reutilizada entre a simulação e o sistema real.

Acknowledgments

The successful completion of this dissertation would not be possible without the help of several people to whom I would like express my utmost gratitude.

First, I would like to thank my supervisors, Helder Fontes and Prof. Manuel Ricardo, for all their support, motivation, and guidance. Their insight and advice were invaluable throughout this project.

A special thanks goes to everyone at the Wireless Networks (WiN) research group of the Centre for Telecommunications and Multimedia (CTM) at INESC TEC. Their suggestions and feedback were greatly appreciated.

I would also like to thank Tom Henderson for his comments and feedback. Furthermore, I have to thank him and all the other developers of ns-3 for their work in creating and maintaining the great network simulator that was used in this project.

Lastly, I am forever grateful to my family and friends for all of their unconditional support and encouragement.

Tiago Bluemel Cardoso

Contents

List of Figures

LIST OF FIGURES

List of Tables

LIST OF TABLES

List of Listings

LIST OF LISTINGS

Abbreviations

AODV	Ad hoc On-Demand Distance Vector Routing
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
CPU	Central Processing Unit
DCE	Direct Code Execution
DCF	Data Collection Framework
IP	Internet Protocol
IPC	Interprocess Communication
INESC TEC	Instituto de Engenharia e Sistemas de Computadores Tecnologia e Ciência
JIT	Just-in-time compilation
MAC	Media Access Control
LSF	Linux Socket Filter
NDlog	Network Datalog
NIC	Network Interface Controller
ns-2	Network Simulator 2
ns-3	Network Simulator 3
NSC	Network Simulation Cradle
OLSR	Optimized Link State Routing Protocol
OS	Operating System
OSPF	Open Shortest Path First
pcap	Packet Capture
RTT	Round-Trip Time
TCP	Transmission Control Protocol
TTL	Time To Live
UDP	User Datagram Protocol
VoIP	Voice over IP
WMRP	Wireless Metropolitan Routing Protocol

Chapter 1

Introduction

The development of new protocols for communication systems generally involves four phases: design, evaluation in a network simulator, evaluation in a real testbed, and deployment.

The first phase consists in the design of a protocol concept that is intended to solve a given problem. From this concept, the researchers can implement a protocol model to be used in simulation. Network simulation is a tool of great value in the evaluation of communication protocols because the variables that influence the network scenario can be easily controlled to create specific and reproducible test conditions. It is also an environment that is easily observable and much cheaper to deploy than a testbed with real hardware. However, evaluation on a testbed must still be done eventually, to test how the protocol behaves in real-world conditions which gives more accurate and credible results. Finally, the protocol can be deployed in the desired systems to solve the problem for which it was initially designed.

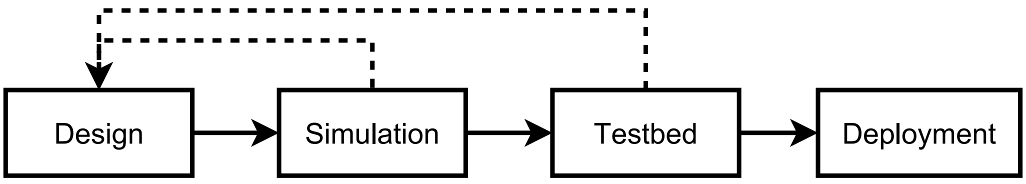

Figure 1.1: Development process of protocols for communication systems

This development process is usually iterative (Figure 1.1). For example, findings in one of the evaluation phases may force the researchers to go back to the design phase and reevaluate some aspects of the protocol. This iteration is necessary to improve the protocol but it is a drawback related to how the simulation model and the protocol testbed implementation are usually developed.

Traditionally, a simulation model of the protocol is created from the concept designed in the first phase. Multiple simulations are run and the results are analyzed and used to tweak and improve the protocol. When the simulation results are acceptable, development can continue to the next phase. A prototype of the protocol is thus implemented in a real system and run in a testbed to be validated. If the results do not meet the expectations then the protocol needs to be changed (first phase) and new simulations made (second phase). At this point, two implementations of the protocol are being maintained: the simulation model and the implementation prototype. This situation leads to duplication of effort when a change needs to be made in the protocol which in turn increases both the development time and the chance of introducing errors in the implementations of the protocol.

Researchers have been trying to find ways to alleviate the problem of having to create and maintain two implementations of protocols that need to be evaluated both in simulation and in testbeds [KLW10]. The solution is **shared protocol implementation**, which consists in developing a single implementation of a protocol that can run both in a simulator and in a real system.

1.1 Motivation

Researchers at the Centre for Telecommunications and Multimedia (CTM) at INESC TEC often need to develop new protocols for communication systems. When doing so, they must go through the development process presented above.

For the simulation phase, researchers at CTM usually use the ns-3 simulator, which is a widely-used open source discrete event networking simulator [ns-a]. Some of its main advantages are the good performance in simulation mode [WvLW09, KBO12] and the fact that it is open source, which allows researchers to share the models they develop and benefit from each others work.

The ns-3 simulator also has features that help users easily extract information from the simulation. The first of these is tracing [ns-d]. Tracing allows model developers to define trace sources in their code that signal events that are happening in simulation and provide the underlying data. Users can then define trace sinks that are connected to sources in order to consume the given data in the way that best serves the necessities of the user. Predefined helpers that can output the tracing data as ASCII or .pcap files are provided. Another of these features is the Data Collection Framework (DCF) which provides capabilities to perform on-line processing of data generated by the simulation [ns-b].

When moving from simulation to testbed, the researchers at CTM also have to deal with the process of porting the simulation code to a real system. A proposed solution to this problem was fast prototyping, a protocol development process in which the simulation code is reused in the testbed with the help of a feature of ns-3 that enables emulation. Emulation allows nodes in a simulation to communicate with the outside world and thus enables them to participate in testbeds. However, it was found that this approach is not scalable to nodes that need to process a large amount of real traffic [CFR11].

The motivation for this dissertation is thus the need for a shared protocol implementation that allows researchers to reuse the same code in ns-3 simulations and in testbeds while at the same time enabling the use of the full functionality provided by the simulator and providing good performance in both the simulation and the real environment.

1.2 Goals

The main goal of this thesis is to propose an architecture for shared protocol implementation that meets the following requirements:

Compatibility with ns-3 The developers must be able to use all the functionality provided by ns-3 (e.g. tracing and DCF) when running the code in simulation. It is acceptable, however, that some ns-3 functionality is unavailable or limited when the code is used in a real environment.

Scalability The implementations of protocols developed with the proposed architecture must perform well in both the simulation and the real system. This means that the performance of the protocol in real systems should not be degraded because of the fact that it must also support being run in a simulator. Conversely, the performance of the simulation should not be degraded because of the fact that the protocol being run is encumbered by details of a real system.

Reusability It should be possible to reuse most of the protocol code between the simulation and the testbed. However, to meet the requirement of scalability, it is acceptable that small parts of the protocol may need to be rewritten.

The proposed architecture should also be validated by implementing already existing protocols and evaluating their performance and the amount of code that was reused.

1.3 Results

In this dissertation we determine that fast prototyping is a good shared protocol implementation solution but it has performance problems in real scenarios because of the overhead introduced by ns-3 emulation.

We therefore propose a solution to the performance problems of ns-3 that consist in migrating the data plane operations of routing protocols to outside of the simulator. Since most of the traffic of typical networks is related to data, this solution improves the emulated nodes efficiency. We present two alternative options to implement this solution: (1) executing the data plane in a userspace process that communicates through interprocess communication (IPC) with the control plane that runs in simulator; (2) executing it in kernelspace by updating the kernel routing tables with the information gathered by the control plane.

In addition to the previous solutions we also propose an architecture that allows applying such solutions to scenarios where multiple nodes need to be emulated in the same machine. This architecture uses Linux network namespaces as lightweight environments where the data plane of each emulated node can run independently of each other.

A new ns-3 module, *real routing*, was also created to implement the data plane in kernelspace architecture. The module allows users to easily use the architecture with their ns-3 protocols.

The proposed architectures were validated by comparing their performance in real environments against traditional emulation as well as the amount of new code that needs to be written to implement them. Results show that significant performance increases can be obtained using the new architectures with only relatively small amounts of new code needing to be developed.

1.4 Document Structure

The remainder of this document is structured as follows. Chapter 2 presents the most relevant state of the art approaches that enable shared protocol implementation with ns-3. Chapter 3 describes the architecture of ns-3 emulation and addresses the performance problems that affect it. Chapter 4 presents the proposed solutions to the performance problems of ns-3 emulation. Chapter 5 contains the methods used to evaluate the proposed solutions and their results. Chapter 6 presents the main conclusions and future work. Finally, appendix A provides the detailed results obtained in the several empirical studies that are presented in this dissertation.

Chapter 2

Approaches to Shared Protocol Implementation

In this chapter we present and analyze the state-of-the-art solutions to shared protocol implementation. The presented solutions employ different approaches in order to allow the same code to be used in both simulation and real systems, but they can be organized in three main categories: abstraction layers, direct code execution, and fast prototyping. For each of these categories, we will describe how it enables shared protocol implementation and present the tools that are currently available.

We conclude this chapter with a comparison between all the presented solutions and with the selection of the approach that will be further developed in order to meet the goals of the project, as stated in Section 1.2.

2.1 Abstraction Layers

An abstraction layer provides a development environment that hides from users the details of the underlying systems on which it is implemented, and can switch between those implementations. If an abstraction layer can be run on top of a simulator and a real system then it can be used for shared protocol implementation (Figure 2.1). Abstraction layers may also support more than one simulator and more than one real system, thus providing even more portability.

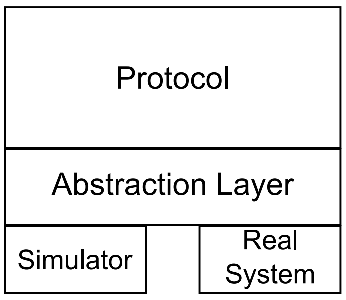

Figure 2.1: Abstraction layer for network protocol development

Developers only have to implement their protocol once using the API provided by the abstraction layer. By switching the implementation used by the layer, the protocol can be run on the simulator or on the real system as needed. Usually, the only thing needed to make the switch is to recompile the code.

The main problem of abstraction layers is that, since they have to support multiple implementations, they cannot provide a very rich API to their users. This happens because different systems offer different features and so the abstraction layer can only provide the features common between all of its implementations. Another possibility is disabling certain features in some implementations or emulating those features at the cost of performance.

In the remainder of this section we will present and analyze some tools that use the abstraction layer approach to provide shared protocol implementation.

2.1.1 RapidNet

RapidNet is a toolkit that enables the development of network protocols for simulation and implementation [MLL+09]. The protocols are specified in a declarative paradigm using Network Datalog (NDlog), a recursive query language. RapidNet can compile the protocol specification into ns-3 code which can then be used for simulation. The same specification can also be used in a real environment by running the ns-3 code in emulation mode (Figure 2.2).

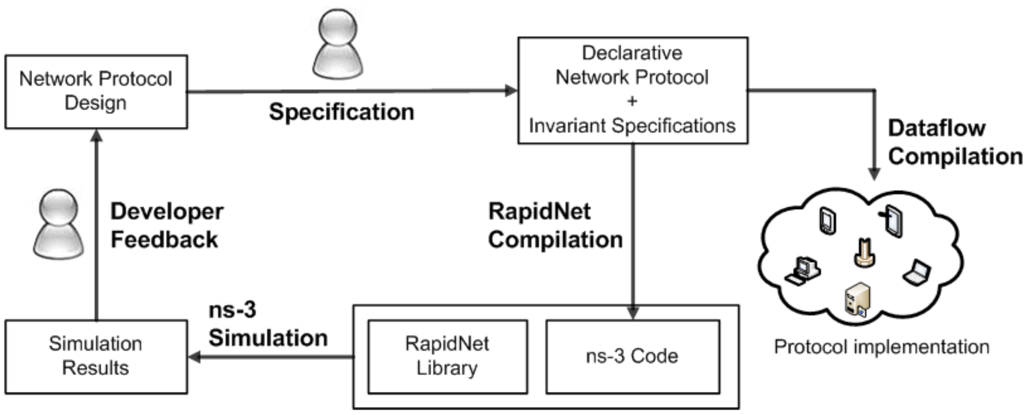

Figure 2.2: The RapidNet development cycle [MLL+09]

One of the problems of RapidNet is that it uses a declarative programming paradigm which is unusual in protocol development and results in a steep learning curve. Another problem is that the generated ns-3 code depends on the RapidNet library an it cannot be submitted for inclusion into ns-3 as a new module.

2.1.2 Protolib

The Protean Protocol Prototyping Library (Protolib) is a toolkit that provides a cross-platform C++ API for the development of network protocols. The developed protocol can run on various systems (including Linux, MacOS, Windows, and FreeBSD among others) and on the ns-2 and OPNET simulators [U.S].

Some interesting classes provided by Protolib are the ProtoSocket, which abstracts the underlying systems TCP and UDP sockets; ProtoTimer, a generic timer class; ProtoRouteMgr, which provides a common interface to the systems routing tables; and ProtoCap, used for raw MAC-layer packet capture.

The main disadvantage of Protolib is that it currently only supports ns-2, not ns-3. However, even if it did, it is unclear if it could provide all the features of ns-3 since some of those features are not available on all systems that Protolib also supports.

2.1.3 Click Modular Router

The Click Modular Router (or just Click) is a flexible software architecture for developing configurable routers [KMC+00]. A router can be created by combining small elements in a graph-like architecture. Each element has a defined number of input and output ports and implements some simple functionality such as decrementing a packet TTL or looking up an IP route. The combination of these elements can create routers with complex functionality providing a flexible platform for researchers to experiment with new protocols (Figure 2.3).

Click can run in several systems such as Linux and FreeBSD and there are also tools that integrate it with ns-2 (nsclick [NJG02]) and ns-3 (ns-3-click [SM11]). The ns-3 simulation can even become more efficient in some cases by using Click to perform the layer 3 functionality, although at the cost of increased memory consumption.

Click also has steep a learning curve since it uses a flow-based programming paradigm [Mor10] which is different from the more usual discrete event simulation used in ns-3. In addition, researchers also have to learn the configuration syntax that is used to describe the connections between the elements of a router.

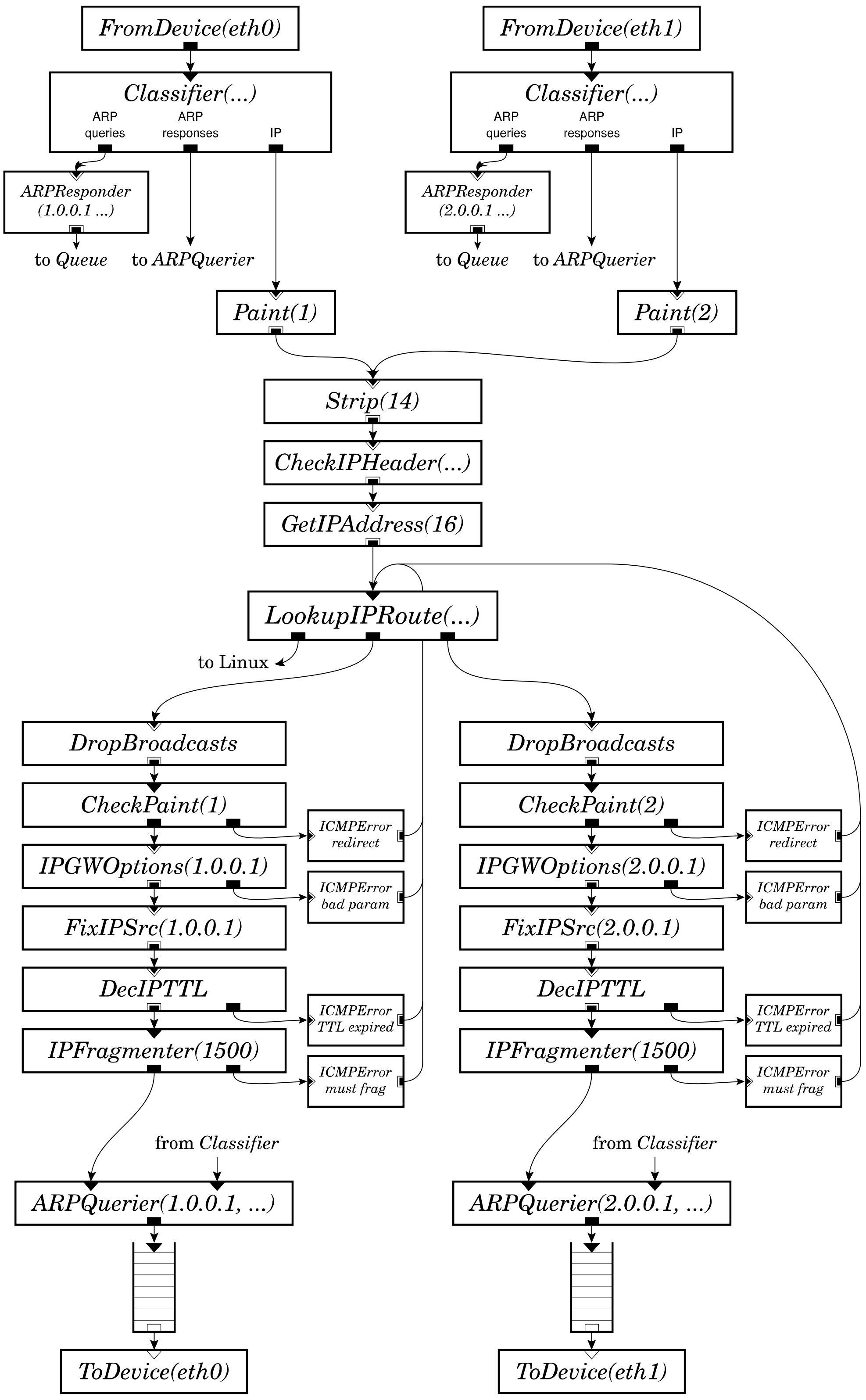

Figure 2.3: Example of a Click IP router configuration [KMC+00]

2.2 Direct Code Execution

Direct code execution (DCE) provides an environment that allows code developed for real systems to run in simulations [Lac10]. This is usually achieved by virtualizing the access to global and static variables and by redirecting the system calls to an implementation for the simulator (Figure 2.4).

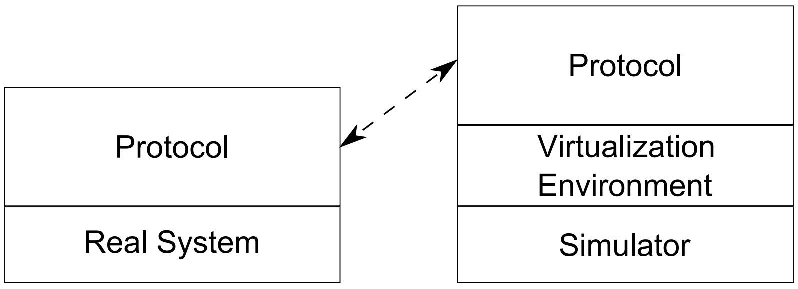

Figure 2.4: Running a real protocol in a simulator using a virtualization layer

The main goal of DCE is to provide a way to run real code in simulations in order to obtain more accurate and realistic results. However, it can also be used as a means to achieve a shared protocol implementation. To do this, the researchers must first develop the code in a real system. They must then use the facilities provided by DCE to run that code in simulation. Afterwards, when they need to run the code in the real system they can just directly use the code they developed without the need of DCE.

Real systems are much more complex and provide many more features than simulators so it may be impossible to provide every functionality of the real system in the simulator. Researchers must be aware of what features are actually implemented by their DCE implementation and may not use any other functions provided by the real systems or else the protocol will not run in the simulator. The converse can also happen, that is, the simulator may provide functionality to the developers that is not available in the real system. In this case the developers also have no way of using those features.

Another problem is that it is usually harder and more time consuming to develop protocols in a real system than in a simulator because the latter offers useful abstractions that facilitate development.

2.2.1 ns-3 DCE

The ns-3 simulator provides a direct code execution framework that is able to execute Linux protocols in simulation without source code changes [TUM+13]. Both userspace and kernelspace protocols or applications can be used, so it is possible, for instance, to use the real ping program or the real Linux networking stack in simulation (Figure 2.5).

Before the development of ns-3 DCE, the Network Simulation Cradle (NSC) [Jan08] was another framework that could execute a kernel stack in ns-3, however it was hard to maintain

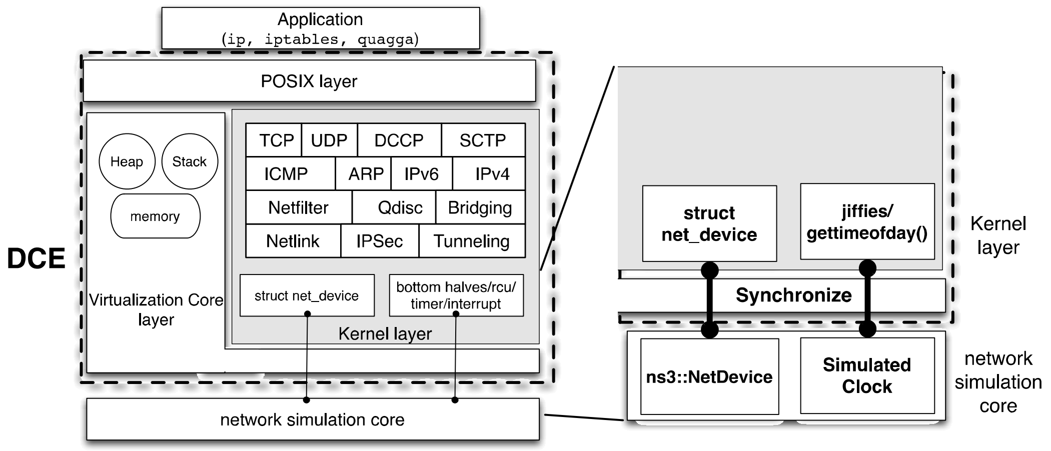

Figure 2.5: Ns-3 DCE architecture [TUM+13]

because it relied on source code transformations. The ns-3 DCE module can however replace most of the functionality provided by NSC [TUT13].

There are a number of problems when using the DCE module as a way to achieve a shared protocol implementation. First, there is a loss of performance in the simulation resulting from having to virtualize the protocol [TUM+13]. It is also more time consuming to develop protocols in Linux than in ns-3 [CFR11], resulting in a longer development time. Finally, not all of the functionality provided by ns-3 can be used since there is no equivalent in Linux. In some cases an alternative is available, for example, it is possible to add tracing support to a real protocol by using aspect based programming [Lac10], but this is not as straightforward as simply using the ns-3 tracing facilities.

2.3 Fast Prototyping

Fast prototyping [CFR11] is a development model in which researchers implement the protocol directly in the simulator. Emulation is then used to allow the simulation code to be reused in testbeds (Figure 2.6).

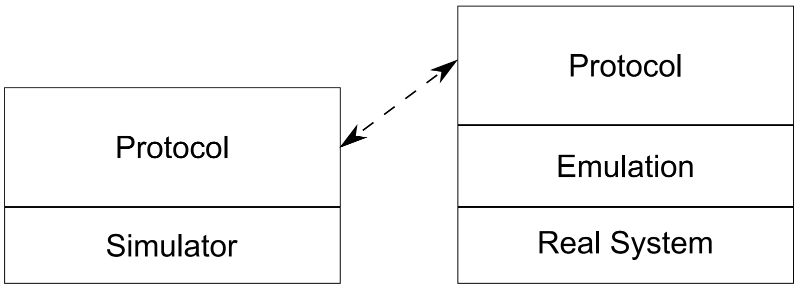

Figure 2.6: Running a simulation protocol in a real system through emulation

The main advantage of fast prototyping is that it allows researchers to use all of the functionality provided by the simulator. It is also generally easier and faster to develop protocols in the simulator since it provides abstractions that facilitate the development.

2.3.1 Simulator-Agnostic ns–3 Applications

Simulator-agnostic ns–3 applications are applications that can run both in simulation and in real systems [AR12]. They are developed as traditional ns-3 applications, but when needed they can communicate with the real environment via real sockets from the underlying system instead of simulated sockets. The actual socket that is used is transparent to the application so its code does not need to be rewritten. This can be done by implementing the real socket functionality as classes derived from the base Socket class of ns-3 and using the factory pattern to create the actual socket as needed (real or simulated). Figure 2.7 shows an example of this for the case of an UDP socket.

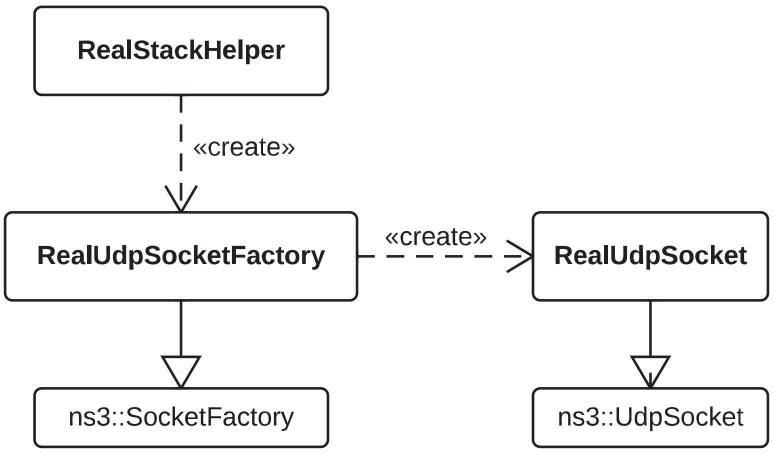

Figure 2.7: Real UDP socket architecture for ns-3 [CFR11]

This approach is very efficient [CFR11] and works well for applications, however it is not enough for the needs of a routing protocol. These protocols need more than simple UDP or TCP sockets, they also need to access and alter routing tables. Many protocols also do not use TCP or UDP, but instead use IP datagrams directly (e.g. OSPF [Moy98]). Others may even work on top of layer 2 protocols (e.g. WMRP [RCF+10]).

Simulator-agnostic ns–3 applications are thus a good complement to other approaches of fast prototyping but on their own they can not offer a complete shared protocol implementation solution.

2.3.2 Ns-3 Emulation

Ns-3 has an emulation module that can be used to integrate simulated nodes into testbeds [ns-c]. In ns-3, the simulated nodes communicate with simulated channels through NetDevices, which are analogous to Network Interface Controllers (NIC) in real systems (Figure 2.8).

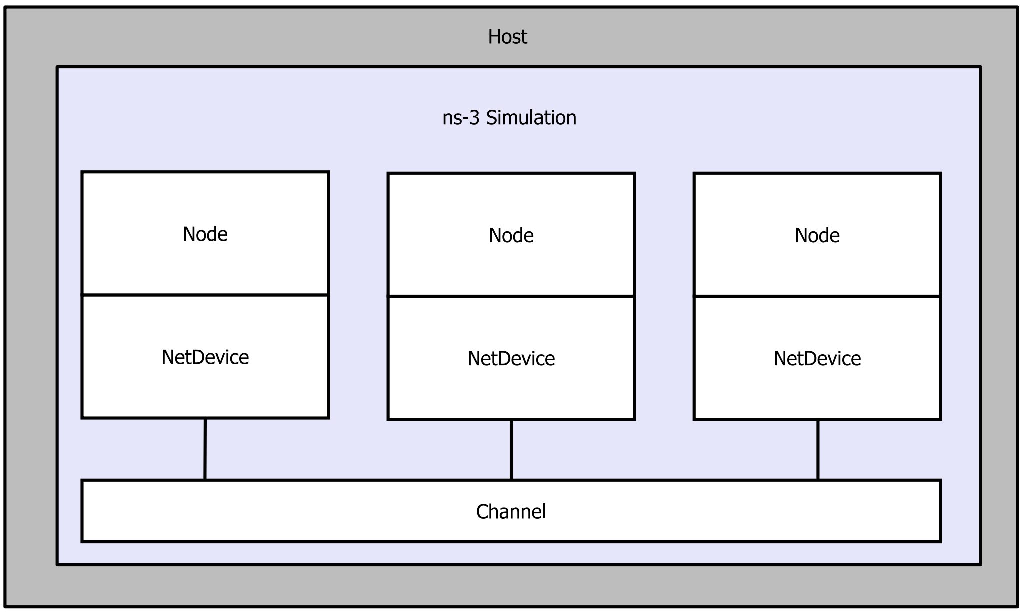

Figure 2.8: Example of a simulation in ns-3 (adapted from [ns-c])

To allow nodes to communicate with the real environment, emulation instantiates the nodes NetDevices as FdNetDevices, which direct their read and write operation to a file descriptor in the real system. A raw socket is then opened in the desired interface of the real system and its file descriptor is assigned to the FdNetDevice. From this point onward, the node is transparently communicating with the real environment (Figure 2.9).

Figures 2.8 and 2.9 show analogous scenarios. The former represents a simulation where three nodes are connected through a simulated channel. The latter shows a testbed scenario where the three nodes now communicate through a real channel. The protocol implementation code for the nodes can be exactly the same in both situations.

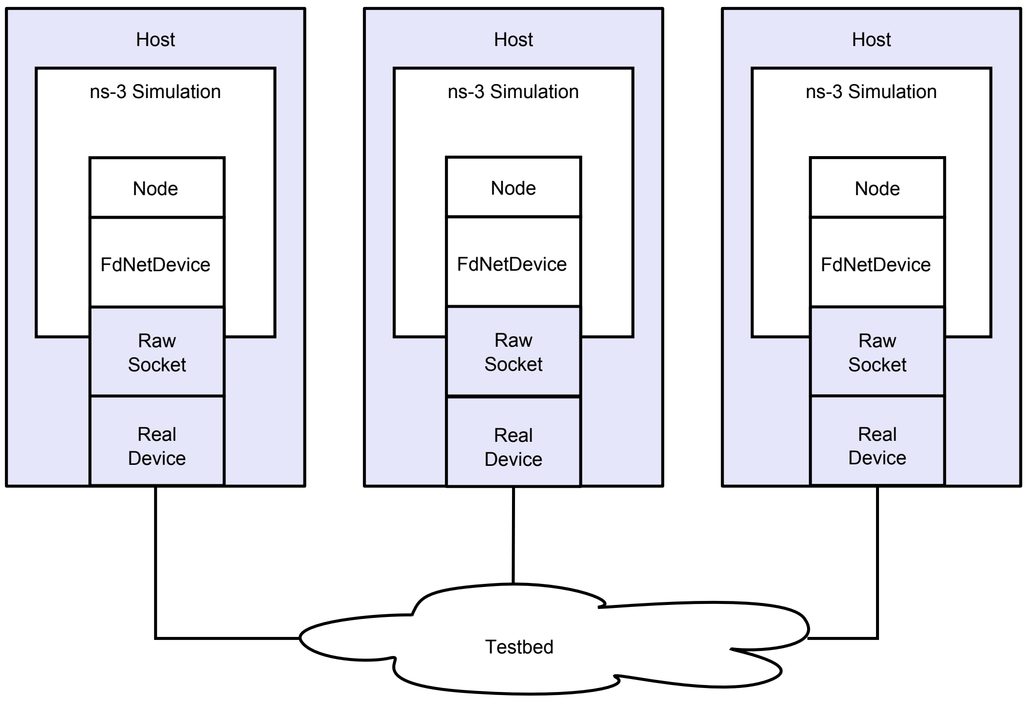

Figure 2.9: Example of emulation in ns-3 [ns-c]

The main disadvantage of ns-3 emulation is that it has poor scalability. The performance of an emulated node quickly degrades as the number of packets it has to process increases [CFR11].

2.4 Comparison and Conclusions

In this chapter we presented three categories of solutions for shared protocol implementation: abstraction layers, which provide a development environment abstracted from both the simulator and the real system; direct code execution, which allows code implemented for real systems to be executed in simulation; and fast prototyping, where code is developed in the simulator and may be reused in a real environment with emulation. Examples of tools that use each of these approaches were also provided.

Table 2.1 shows a comparison between the different tools presented. We are interested in whether the tools achieve the goals stated in Section 1.2, namely the goal of compatibility with ns-3 (allowing to use all of its features) and scalability in simulation and in testbeds.

Table 2.1: Comparison of different approaches to shared protocol implementation

Approach	**Compatibility with ns-3**	**Scalability in simulation**	**Scalability in testbed**
RapidNet	High	Medium	Medium
Protolib	None	N/A	High
Click	Medium	Medium	High
ns-3 DCE	High	Low	High
Agnostic Apps	Full	High	High
ns-3 emulation	Full	High	Low

First, RapidNet has a good compatibility with ns-3 since its integration with the simulator was part of its initial design. However, it cannot achieve the same level of scalability as other solutions mostly because of the mismatch between its declarative development environment and the systems on which it must run.

Protolib has no support for ns-3 yet, so it cannot be used directly. It might be possible, however, to run Protolib protocols in ns-3 by compiling them to Linux and then running the executable with DCE. This solution would suffer from the same problems that DCE has however. The upside is that is easier to develop protocols with Protolib than in a real system because Protolib provides higher level abstractions.

The Click Modular Router can be integrated with ns-3 through ns-3-click but this approach only supports the development of layer 3 protocols. Using Click can also reduce the scalability of the simulation due to increased memory usage.

The ns-3 DCE framework allows the execution of real code in the simulator. Because this framework is part of ns-3 it tries to provide a high level of compatibility with it, however, it cannot provide access to the full features of ns-3. Since the protocol code is created for a real system it has high scalability in testbeds, however the performance suffers when used in simulation because of the need for the virtualization layer.

Agnostic applications are developed directly in ns-3 so they can leverage the full functionality of ns-3 while providing high scalability in simulation. In a testbed, this approach provides high scalability if the application only uses UDP or TCP sockets. The problem is that network protocols also need to ensure other traffic is forwarded to the correct destination, so it is not possible to use this approach.

Finally, using emulation it is also possible to use all the features provided by ns-3 and have high scalability in simulation because the protocols are developed directitly in ns-3. However, this approach has low scalability in a real environment because emulation introduces overhead in the processing of each packet.

We have analyzed different tools in the three main categories of shared protocol implementation: abstraction layers, direct code execution, and fast prototyping. In the first category development is done in an environment that abstracts the underlying systems while in the second development is done in a real system. Since in both of these approaches the protocols are not developed directly in the simulator, they cannot provide the full functionality of ns-3. Because of this, both abstraction layers and direct code execution do not achieve the goal stated for this project of full compatibility with ns-3 when the protocol is used in simulation. Only with fast prototyping, in which the code is developed directly in ns-3, can this goal be achieved.

In the fast prototyping category we have agnostic applications, which are a good solution, but we can not use them because they are only directed towards the implementation of simple applications and not for routing protocols. We are left with ns-3 emulation which achieves most of the stated goals, including reusability, full compatibility with ns-3 and scalability in simulation. The only goal missing is scalability in a real environment.

Therefore, we have to improve the performance of ns-3 emulation in order to have a fully functional fast prototyping solution for shared protocol implementation that meets the goals stated for this project.

Chapter 3

Performance Problems of ns-3 Emulation

In this chapter we present the performance problems that prevent ns-3 emulation to be used in real environments with a large amount of traffic. We first describe the mechanisms used by ns-3 that allow simulated nodes to communicate with real networks and explain why they introduce overhead in the processing of real packets. We then present an empirical study that shows how this overhead affects the performance of ns-3 when it is used to forward a large quantity of packets.

3.1 ns-3 Emulation Architecture

One of the many classes present in ns-3 is the NetDevice, which provides simulated nodes with an interface to the MAC layer and is mainly used by layer 3 protocols to send and receive packets. Some of its subclasses are the CsmaNetDevice and WifiNetDevice, for example. To provide emulation functionality, ns-3 defines the FdNetDevice, another subclass that implements the NetDevice interface by reading and writing packets to a file descriptor in the real system.

In Linux, emulation can be achieved by opening a raw packet socket (a type of socket that can send and receive L2 frames) on the desired real network interface and instantiating an FdNetDevice object with the file descriptor of the created socket. The simulated node that uses this NetDevice will then transparently communicate with the real network.

With this architecture, the process of sending packets to the network is straightforward. When a node requests a packet to be sent (using the *Send* or *SendFrom* functions of the NetDevice interface) the FdNetDevice implementation simply has to issue a *write* system call on the socket file descriptor with a buffer containing the contents of the packet. Converting an ns3::Packet to a memory buffer is trivial because ns-3 already stores packets in the network format. This design decision was made specifically to facilitate emulation [Lac10]. However, a buffer still needs to be

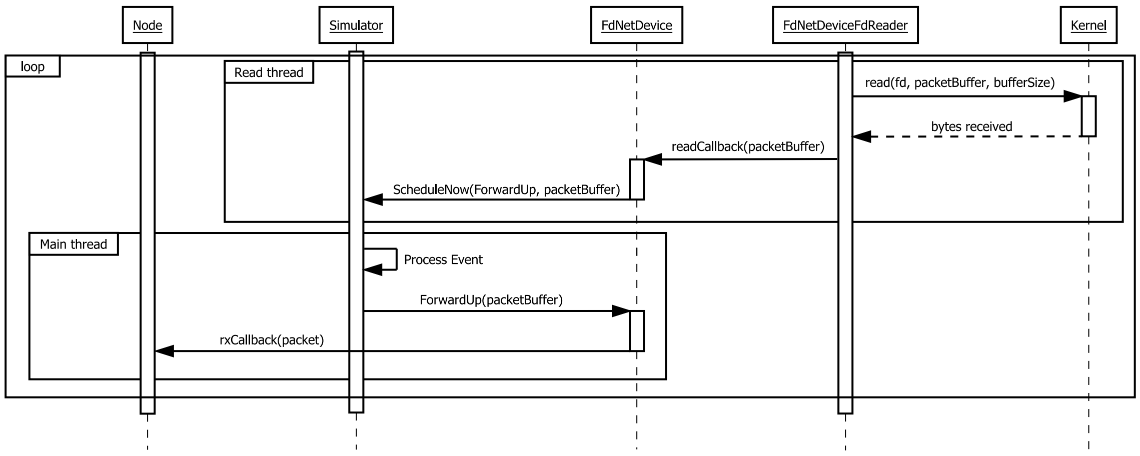

Figure 3.1: Reading network packets with FdNetDevice

dynamically allocated and the packet data copied to it in order to perform the write. This process introduces some overhead that may affect performance if many packets need to be sent.

The process to receive network packets is more complex (Figure 3.1 shows an overview). Since the simulation event loop cannot be blocked while waiting for packets to arrive, the reading is, instead, done in a dedicated thread. Each FdNetDevice creates an FdNetDeviceFdReader which starts a new thread and then initiates a reading loop where it issues a blocking *read* system call on the socket file descriptor. When the *read* call returns, the FdNetDeviceFdReader passes the buffer containing the received packet to the FdNetDevice. The FdNetDevice cannot deliver the packet to the node yet because a simulation event must first be generated. To do that, the event is scheduled with the simulator. The simulator must then synchronize with the main thread (resulting in some overhead) in order to process the event. It then calls FdNetDevice::ForwardUp with the packet buffer. The FdNetDevice can now create an ns3::Packet from the buffer. As described above, this is a simple process but it does require some space to be dynamically allocated and the buffer to be copied there. Again, this process adds to overhead of packet reception. The newly created ns3::Packet is then finally delivered to the simulated node.

The approach taken by ns-3 to emulation allows for any network operation that works above the MAC layer to transparently communicate with the real environment. As we showed, however, the implementation introduces some overhead to the processing of network packets.

3.2 Performance Evaluation of ns-3 Emulation

We performed an empirical study to demonstrate the overhead introduced by ns-3 emulation in the processing of real network traffic. In this experiment we compare the performance of a Linux bridge when packet forwarding is done in kernelspace, userspace, or using ns-3 emulation.

The three scenarios have the same network topology (Figure 3.2), which is composed of three nodes connected by 100 Mbit/s Ethernet links:

- The source **S**. It uses the `iperf` tool to generate UDP traffic destined to **D**. At the same time it also pings **D** to measure the round-trip time (RTT);
- The destination **D**. It runs an `iperf` server to receive the UDP traffic from **S** and calculate the throughput and packet loss;
- The bridge **B**. It forwards the frames between **S** and **D**.

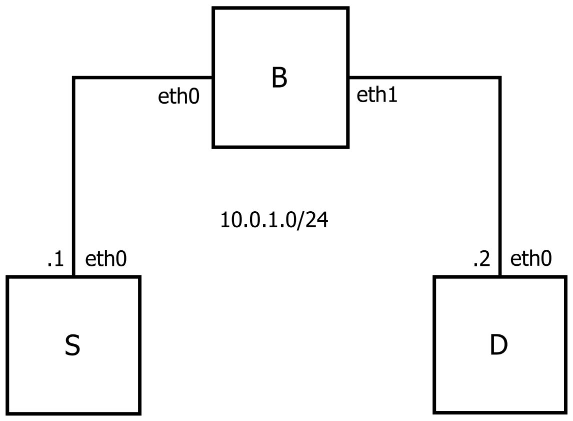

Figure 3.2: Network topology of the test scenarios

The machine used to act as the bridge runs on an Intel Atom N270, which is a single core processor but that uses the Hyper-Threading technology so it behaves as having two logical cores. The operating system used was Ubuntu 14.04 LTS.

There were three scenarios tested. In the first scenario **B** is configured to forward packets using a kernelspace Linux bridge connecting eth0 and eth1. In the second scenario forwarding is done by an userspace process that reads the frames from eth0 and writes them to eth1 and vice versa. Lastly, in the third scenario **B** runs an ns-3 simulation that uses emulation to connect to eth0 and eth1 and bridge the traffic between them.

For each scenario, two series of tests were executed, each starting with **S** generating 1 Mbit/s of UDP traffic to **D** and then increasing that value to 2, 4, 8, 16, 32, 64, and 90 Mbit/s. Each test was repeated 5 times and had a duration of 30 seconds. The first series of tests had a UDP payload of 1470 bytes, which is representative of typical application traffic. In the second series of tests, the payload of the UDP packets was 160 bytes. This small size, which is usual in VoIP traffic, was used to generate a large number of packets in order to show the performance decline in ns-3 when it has to process an increasing number of packets. For these tests, the last offered data rate was of 70.8 Mbit/s instead of 90 Mbit/s because that is the maximum throughput that can be achieved in 100 Mbit/s Ethernet links for UDP packets with 160 bytes of data.

For each of these tests, four performance metrics were measured:

- The received data rate in **D**, which was given by the `iperf` tool;
- The packet loss ratio, which was also measured by `iperf`;

- The average round round-trip time, measured using the `ping` utility;
- The average CPU load in **B**. This was measured using the `time` utility for the userspace and ns-3 scenarios. For the kernelspace scenario the utility `top` was used instead because there is no single process that `time` can measure.

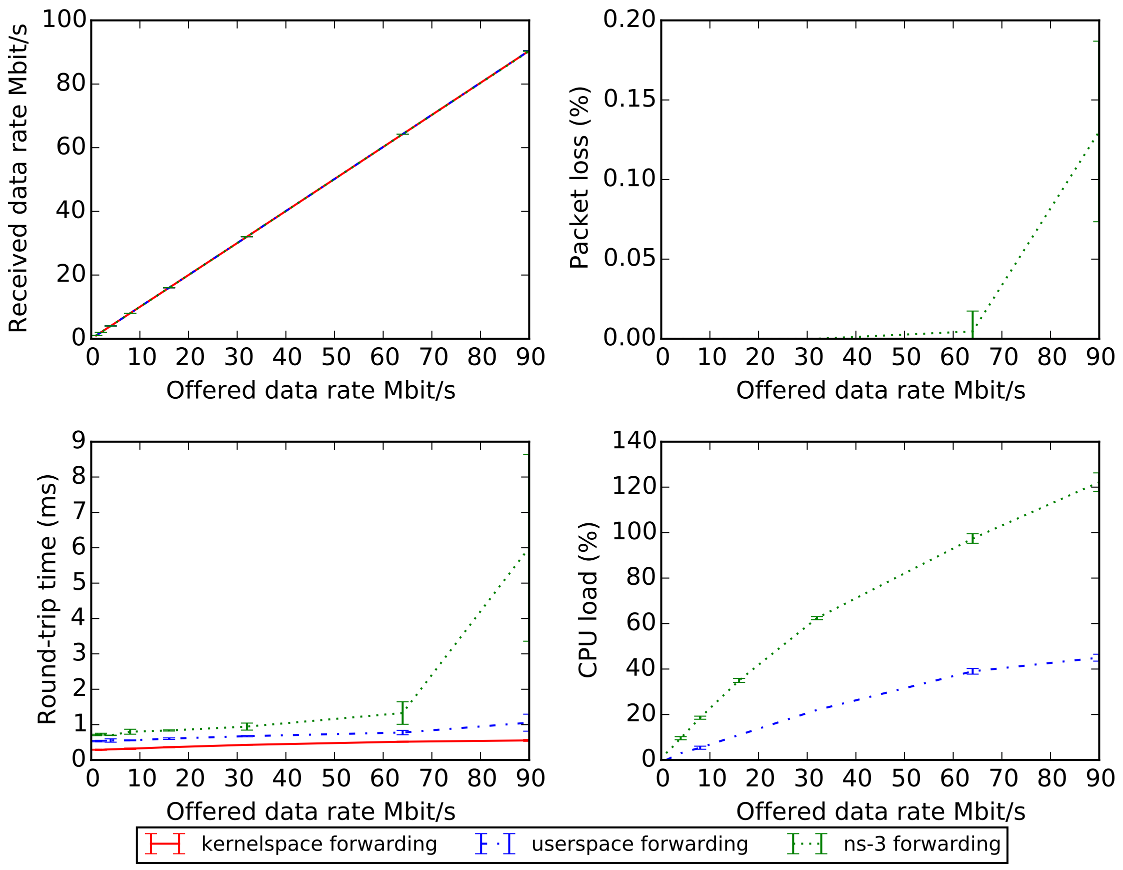

Figure 3.3: Forwarding test results for UDP packets with payload of 1470 bytes

The results for packets with a payload of 1470 bytes are presented in Figure 3.3. For packets of this size, all implementations are able to forward the generated traffic at any data rate with minimal packet loss. A small difference can be found in the RTT measures due to the userspace processing of packets and the other operations described above that ns-3 performs for each packet. The larger difference is found on the CPU load on the bridge. While the load imposed by the kernel implementation is negligible, the load in the userspace scenario increases proportionally to the offered data rate until it reaches approximately 40% for 90 Mbit/s. The CPU load in the ns-3 scenario increases at a faster rate reaching 120% when the offered rate is 90 Mbit/s. The reason why ns-3 goes above 100% load is that the CPU of the bridge has two logical cores (due to Hyper-Threading) and ns-3 uses a dedicated thread for reading operations.

The results for packets with a payload of 160 bytes are shown in Figure 3.4. The kernelspace implementation is able to forward packets at the offered rate while incurring negligible packet

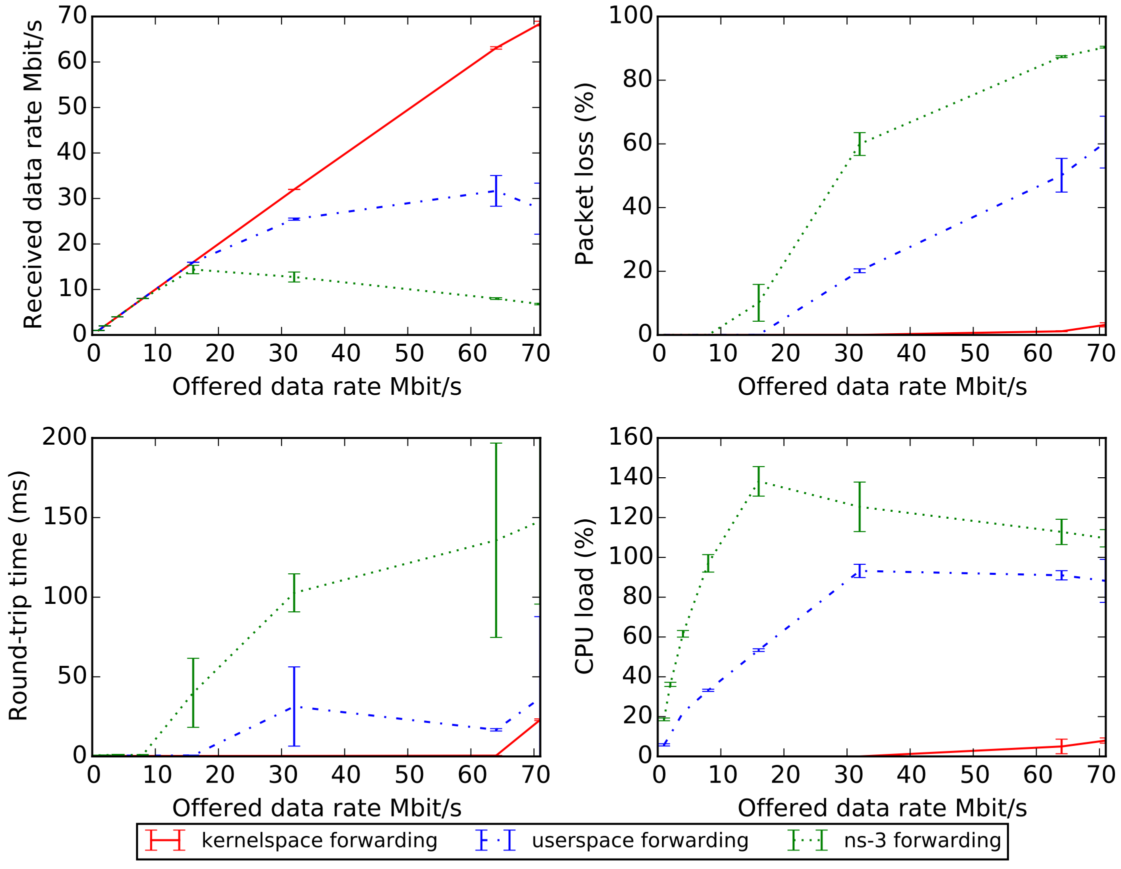

Figure 3.4: Forwarding test results for UDP packets with payload of 160 bytes

loss and CPU load. Additionally, the RTT is always very low (less than 0.5 ms) except when the offered data rate reaches 90 Mbit/s where it increases to around 23 ms.

The forwarding performance of the ns-3 implementation starts degrading just after the traffic generated by **S** hits 16 Mbit/s. When the offered data rate reaches 64 Mbit/s, ns-3 can only forward at a rate of around 8 Mbit/s, less than a third of what the userspace implementation achieves. At this rate there is a packet loss of almost 90% for the ns-3 scenario and half of that for the userspace scenario. As for the RTT, the userspace implementation never goes over 50 ms in the worst case whereas ns-3 reaches 150 ms. Finally, both the ns-3 and the userspace implementation increase their CPU load linearly with the offered data rate, however the load imposed by ns-3 increases at a faster rate.

3.3 Conclusions

Ns-3 provides an emulation mechanism that allows simulated nodes to transparently communicate in real networks. However, the approach used by ns-3 introduces overhead in the processing of each packet. The main causes of this overhead are the context switch between the kernel and user space, the dynamic allocations and copies needed to create ns3::Packets from the data buffers, and the synchronization needed between the read and main threads in order to schedule the reception of received packets.

Despite this overhead, ns-3 emulation manages to perform well in networks where the traffic does not consist of a large number of packets, even if those packets are big in size. The problem manifests itself when ns-3 needs to process a large amount of packets, even if they are small.

A more scalable solution is thus needed to enable researchers to reuse their ns-3 protocol code in real environments where nodes have to process a large quantity of packets at a high bitrate.

Chapter 4

Data Plane Outside of Simulation

This chapter presents an architecture for shared protocol implementation that improves the scalability of ns-3 emulation allowing researchers to reuse most of their simulation code in testbeds with large amounts of traffic.

In a network node, there are two planes of operation: the control plane and the data plane. The control plane is responsible for discovering and maintaining network routes and ensuring connectivity. The data plane uses the routing information generated by the control plane to forward network packets. In a well designed network, most of the traffic corresponds to data and, as was shown in Section 3.2, processing large amounts of packets in ns-3 is less efficient than doing it in userspace or kernelspace. Thus, by moving the data plane operations outside of simulation the overhead associated with this kind of traffic can be greatly reduced.

We propose two alternatives to doing this: executing the data plane in userspace and executing it in kernelspace. These approaches are described in the next sections.

4.1 Data Plane in Userspace

The first approach consists in running the data plane outside of ns-3 in userspace. The control plane is still executed in ns-3 and communicates with the real environment through emulation, but the data plane is executed in a userspace process to avoid the overhead of processing the large amount of data traffic in the simulator (Figure 4.1).

The two planes are executing in different processes so interprocess communication (IPC) is needed to allow them to exchange information. For example, the control plane needs to update the routing table information in the data plane so that data packets can be forwarded correctly. The data plane may also need to send feedback to the control plane or request a route in the case of reactive protocols.

The classification of the packets that should be delivered to the control and data planes is done through filters applied to the raw sockets using the Linux Socket Filter (LSF) facility [Ins01].

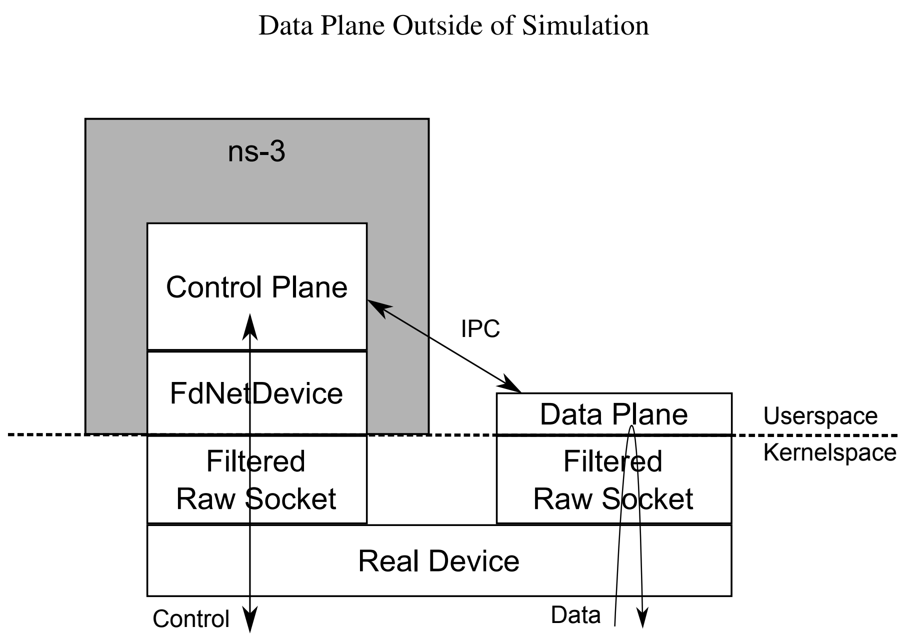

Figure 4.1: Data plane in userspace

Listing 4.1 shows an example filter that accepts only IPv4 packets. At line 0 we load the half word at position 12 in the frame (the EtherType field). Line 1 compares the previously loaded half word to 0x800 (the IPv4 EtherType). If both are equal we jump to line 2 which returns 65535. This will result in the first 65535 bytes of the frame being accepted and delivered to the userspace process. If the comparison of line 2 fails, we jump to line 3 which returns 0. This means that no bytes of the frame are accepted and so it is dropped.

A simple way to obtain the code of socket filters is to run `tcpdump` with the `-d` option. For example, running `tcpdump -i eth0 -d ip` will return the code in Listing 4.1.

```
0  ldh      [12]
1  jeq      #0x800          jt 2   jf 3
2  ret      #65535
3  ret      #0
```

Listing 4.1: Linux Socket Filter code that only accepts IP packets

Socket filters are configured in userspace but they are executed by the kernel on received frames so they are very efficient because packets not accepted by the filter are not delivered to userspace. This way, the control and data planes only receive their respective traffic, avoiding the overhead caused by the processing of unneeded packets.

Using this architecture, the control plane code can be reused between simulation and testbed but the data plane code needs to be rewritten for the real system. However, because the data plane is usually much simpler than the control plane, only a relatively small amount of code needs to be ported.

4.2 Data Plane in Kernelspace

It is possible to specialize the previous architecture for the case of L3 routing protocols. For protocols of this type, whose objective is to forward IP packets, we can update the kernel routing tables with the information gathered by the control plane. This way the kernel will be forwarding the traffic it receives (Figure 4.2) which is more efficient than doing it in userspace (see Section 3.2).

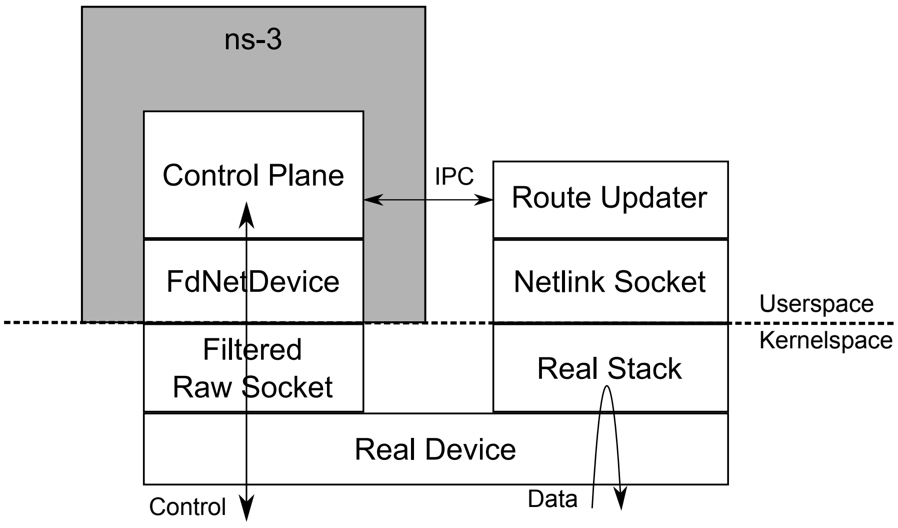

Figure 4.2: Data plane in kernelspace

In this architecture, a route updater process receives the routing information from the control plane and updates the kernel routing tables with that information. To do this, the route updater process uses a netlink socket [SKKK03]. Netlink is a communication channel between kernelspace and userspace and, among other functionalities, allows user processes to update and retrieve routing information. Root access is needed to use this functionality so, in order to avoid having to run the whole simulation as root, the route updater is executed in its own process as root and communicates with ns-3 trough IPC to receive the routes generated by the control plane.

As in the previous architecture, the raw socket used by the control plane needs to be filtered to avoid the overhead of processing unnecessary network traffic.

This approach is more efficient than the previous one since the forwarding is done in the kernel instead of userspace. In addition, there is even less code that needs to be rewritten because the route updater code can be reused by all the protocols that use this architecture. The downside is that this approach is only applicable to proactive L3 protocols, where the control plane is more independent from the data plane, as the kernel may not provide the feedback needed. For example, reactive protocols such as AODV need to know if a route was not found for a given packet in order to initiate the route request procedure [PBRD03]. To support this type of protocols the proposed architecture could be extended with a kernel module that listens on the desired events and reports them back to the control plane.

4.3 Emulating Multiple Nodes

The two architectures previously proposed (data plane in usersapce and data plane in kernelspace) assume that only a single node is being emulated by each machine of the testbed. However, it is many times desirable to emulate multiple nodes (arranged in a given topology) in a single machine. Figure 4.3 shows an example of a single host emulating three ns-3 nodes.

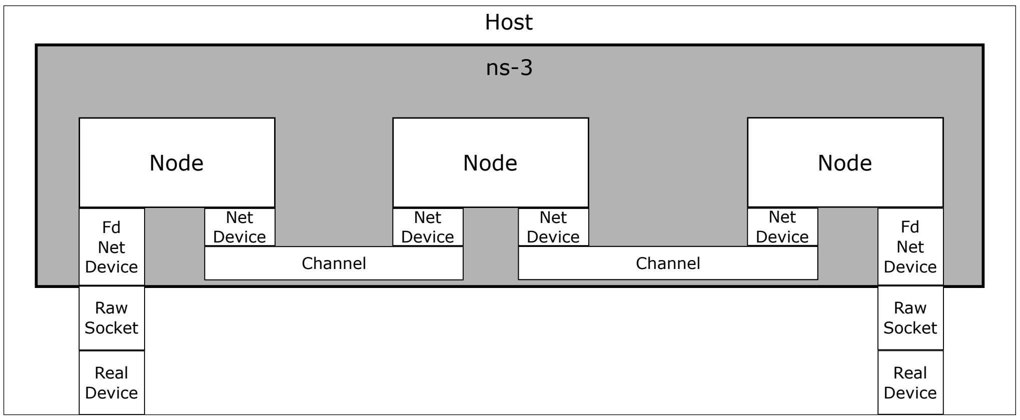

Figure 4.3: Emulation of multiple ns-3 nodes in a single host

To apply the previous architectures to emulations of multiple nodes, we need to have execution environments in the host system where the data plane of each node will run. These environments must have the following properties:

1. Be independent of each other. For example, adding routes to the routing table of an environment must not affect the routing tables of the other environments;
2. Be able to forward the data traffic between each other and with the nodes outside the system;
3. Be efficient. They must not introduce more overhead than what is saved by moving the data plane to outside of ns-3.

The Linux operating system provides environments with these capabilities called network namespaces [Edg14]. These namespaces allow the network resources of the system to be isolated into different independent environments. Each network namespace has its own set of devices, addresses, ports, routing tables, firewall rules, etc.

For an efficient emulation architecture, network namespaces are more appropriate than virtual machines or Linux containers because they only isolate what is needed (the network resources), and so are more lightweight than the other virtualization solutions.

The topology of the simulated nodes in ns-3 must be replicated outside by the network namespaces. This is done by creating a network namespace for each node and connecting them in the same way they are connected in the simulator. The network namespaces also need to be connected to each other in a way that replicates the connections between the simulation nodes. To do that, virtual network devices (`veth`) are used. These virtual devices are created in pairs and assigned to two namespaces that need to be connected. What is written in one `veth` can be read in the other and vice-versa. This way, the network namespaces can forward the data traffic between each other.

The virtual devices can also be configured to match the characteristics of the simulated node (e.g. delay, bitrate, packet loss). This is possible by using the netem kernel module [The09]. For example, to set the delay of virtual device `veth1` to 100 ms, we can use the command `tc qdisc add dev veth1 root netem delay 100ms`.

Figure 4.4 shows how the data plane in kernelspace architecture can be applied to the example in Figure 4.3 by using network namespaces and virtual devices.

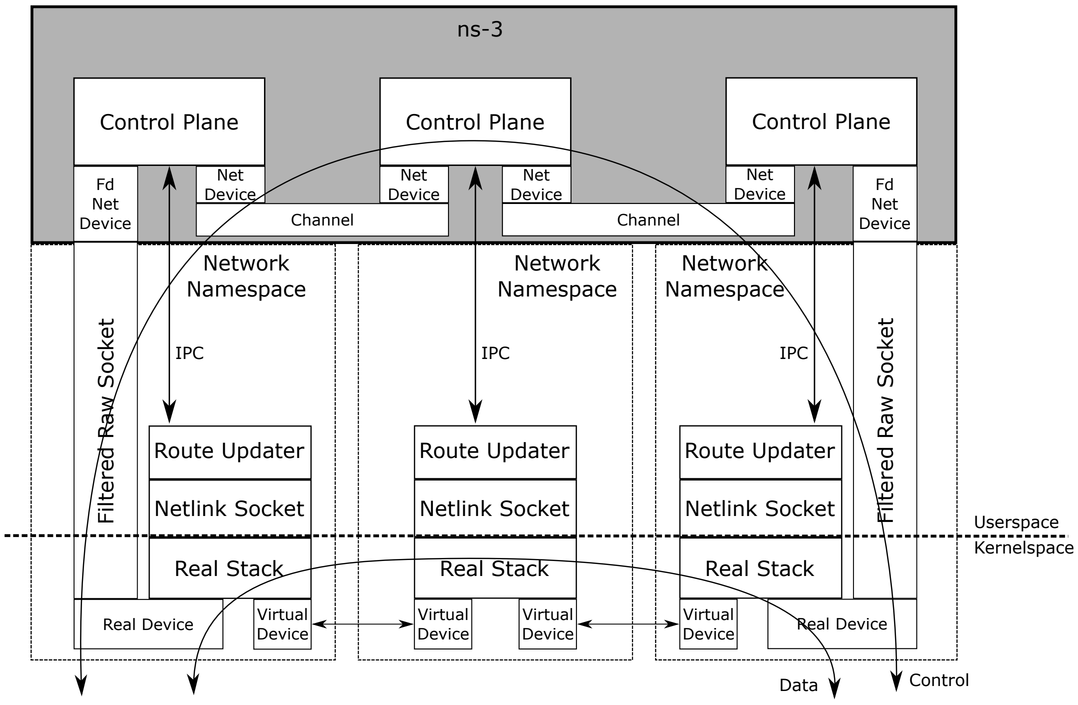

Figure 4.4: Data plane in kernelspace for emulation with multiple nodes

For each ns-3 node in the emulation, a network namespace is created and a route updater for that node is executed in that namespace. This way each node has its own set of routing tables in the kernel that is independent of the other nodes. To connect the nodes, virtual network device pairs are created and each `veth` is assigned to a network namespace and configured to match the corresponding net device of the simulation (e.g. IP address, subnet mask, or emulation characteristics that can be set with netem such as delay and packet loss). The real devices of the host are also moved to the network namespace of the nodes that should be connected to the real network. The filters of the raw sockets are applied in the same way as in the case of emulations of only one node.

The control traffic is thus processed completely within ns-3 while the data traffic never leaves kernelspace. Data packets arrive in a real device, are processed by the real stack of a network namespace, and are forwarded to the next namespace through a virtual device. This process continues until the data packets leave the host through a real device.

Chapter 5

Architecture Validation

In this chapter we evaluate the proposed solutions to make sure that they achieve the goals stated in Section 1.2, namely: compatibility with ns-3, scalability in simulation and real systems, and code reuse.

There is no need to validate the compatibility with ns-3 and scalability in simulation because the protocols are developed directly in ns-3 and so are as compatible and scalable as any other ns-3 protocol as long as they are correctly implemented. Since the proposed changes to network protocols are only needed in testbeds, our validation focus on scalability in real environments and in the amount of new code that needs to be developed to use the proposed architectures.

The scalability in real environments requirement was validated by selecting routing protocols already implemented in ns-3 and extending them to use the new architectures. The performance of both implementations was then compared in a real scenario.

The requirement of code reuse was validated by analyzing the number of lines of code that were added to the implementations of the protocols in ns-3 in order to make them use the proposed architectures.

5.1 Data Plane in Userspace

5.1.1 Performance Validation

To validate the data plane in userspace architecture we used the Wireless Metropolitan Routing Protocol (WMRP), an ad hoc, proactive routing protocol that operates over layer 2 [RCF$^+$10].

The scalability in real environments will be validated by comparing the performance of WMRP running completely in ns-3 emulation with the performance of WMRP when the data plane is executing in userspace. We then analyze the performance gains that were obtained with the new architecture.

The two scenarios have the same network topology (Figure 5.1), which is composed of four nodes connected by 100 Mbit/s Ethernet links:

- The source **S**. It uses the `iperf` tool to generate UDP traffic destined to **D**. At the same time it also pings **D** to measure the round-trip time (RTT);
- The destination **D**. It runs an `iperf` server to receive the UDP traffic from **S** and calculate the throughput and packet loss;
- The routing bridges **RB1** and **RB2**. They forward the frames between **S** and **D**.

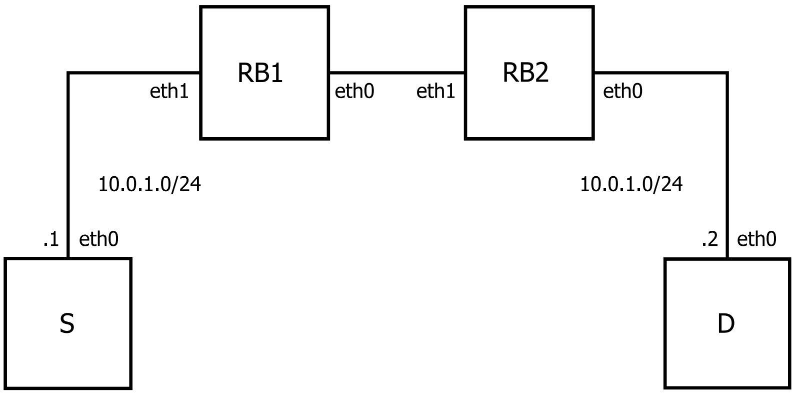

Figure 5.1: Network topology of the test scenarios

There were two scenarios tested. In the first scenario (*ns-3 emulation*), the two RBridges are implemented using tradtional ns-3 emulation. In the second scenario (*userspace forwarding*), the proposed architecture for executing the data plane in userspace is implemented in both RBridges.

For each scenario a series of tests was executed, starting with **S** generating 1 Mbit/s of UDP traffic to **D** and then increasing that value to 2, 4, 8, 16, 32, 64, and 70.8 Mbit/s. Each test had a duration of 30 seconds and was repeated 5 times. The payload of the UDP packets was 160 bytes. The maximum offered data rate was of 70.8 Mbit/s because that is the maximum throughput that can be achieved in 100 Mbit/s Ethernet links for UDP packets with 160 bytes of data.

For each of these tests, four performance metrics were measured:

- The received data rate in **D**, which was given by the `iperf` tool;
- The packet loss ratio, which was also measured by `iperf`;
- The average round round-trip time, measured using the `ping` utility;
- The average CPU load in **RB2**, measured using the `top` utility.

The machine running **RB2** has an Intel Atom N270, which is a single core processor but that uses the Hyper-Threading technology so it behaves as having two logical cores. The operating system used was Ubuntu 14.04 LTS.

The results are shown in Figure 5.2. The traditional ns-3 emulation implementation is able to forward packets at the offered rate until 8 Mbit/s. After that, performance starts degrading and when the offered rate hits 70.8 Mbit/s, ns-3 emulation has 87% packet loss, can only forward at

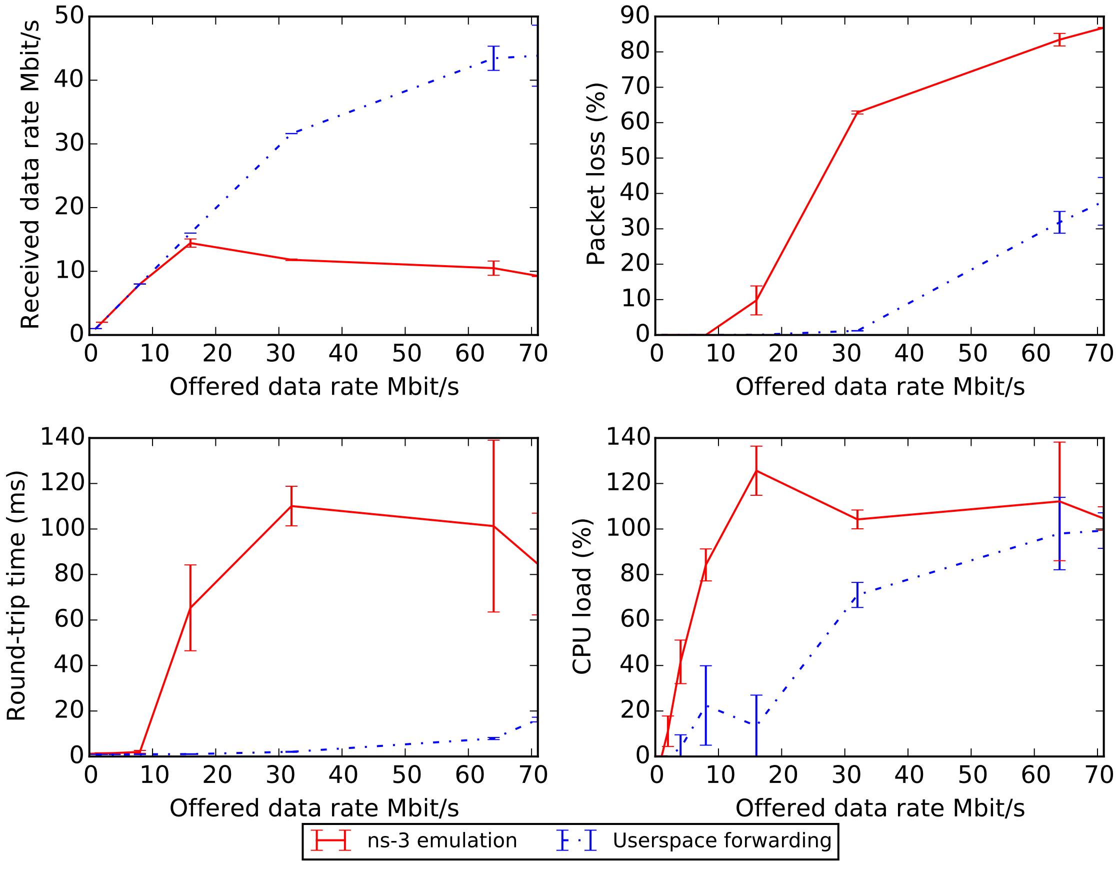

Figure 5.2: Data plane in userspace performance validation results for UDP traffic with payload of 160 bytes

9 Mbit/s, and the round trip time (RTT) also increases to 85 ms. The reason why ns-3 emulation goes above 100% load is that the CPU of the bridge has two logical cores (due to Hyper-Threading) and ns-3 uses a dedicated thread for reading operations.

The data plane in userspace implementation is able to forward packets at the offered rate until 32 Mbit/s while keeping RTT low (1 ms until 16 Mbit/s and 2 ms at 32 Mbit/s). For higher data rates performance starts degrading but at 70.8 Mbit/s, it can forward at around 44 Mbit/s (4.9 times more than traditional emulation) while keeping RTT at 16 ms (5.3 times lower than traditional emulation). Additionally, while the offered data rate is lower than 64 Mbit/s the CPU load of the data plane in userspace implementation is lower than that of ns-3 emulation. For higher data rates both implementations present similar loads, but the userspace implementation is processing more traffic for that same load.

5.1.2 Code Reuse Validation

To validate that using the data plane in userspace architecture does not require writing a large amount of new code we counted the lines of code that were developed to implement the proposed architecture on top of the existing ns-3 implementation of WMRP.

The results are presented in Figure 5.3, where *ns-3 emulation* indicates the number of lines of code of the existing ns-3 implementation of WMRP, and *userspace forwarding* indicates the sum of those lines with the new lines that were developed to implement the data plane in userspace. The blank and comment lines are not counted.

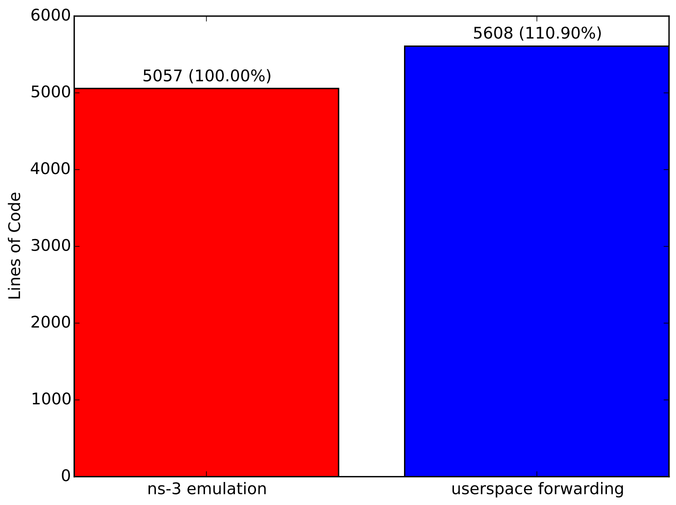

Figure 5.3: Code reuse validation results for data plane in userspace

It can be seen that only around 11% more lines of code were written to implement the data plane of WMRP in userspace. This is in part because the implemented data plane was simplified and assumes that the topology of the network does not change. Implementing the full operations of the data plane would increase the number of lines of code but in general, the data plane is much simpler than the control plane so only a relatively small amount of code needs to be ported.

Ultimately, it is the researchers developing the protocol who need to decide if the extra amount of code that needs to be written is worth the performance gains that will be obtained.

5.2 Data Plane in Kernelspace

5.2.1 *Real Routing* Module

Most of the data plane in kernelspace architecture can stay the same across different protocols, namely the route updater and the interprocess communication. The only part that is specific to each protocol is the socket filter used. This is unlike the data plane in userspace architecture, which requires the data plane (which can be very different between protocols) to be ported to the real system. To enable researchers to quickly use the data plane in kernelspace architecture, an implementation was developed for ns-3, the *real routing* module.

The *real routing* module implements a route updater (*RtNetlinkRouteUpdater*) that uses Netlink sockets to update the kernel routing tables and that runs in a privileged userspace process to avoid having to execute the whole simulation with root privileges. Inside ns-3, the class *RealRouteUpdater* was created to spawn and configure the *RtNetlinkRouteUpdater* process, receive the routes from the routing protocols and send them to the created process, which in turn updates the kernel. The interprocess communication, used to send the routes from the simulator to the route updater, is implemented with Unix domain sockets.

In order for ns-3 routing protocols be able to update the routes of the system without having to be coupled to the new *real routing* module, three new optional callbacks were added to *Ipv4RoutingProtocol*, the base class of all IPv4 protocols (support for IPv6 was not created but it can be easily added when needed). The new callbacks are: RouteAddedCallback (called when a the protocol creates a new route), RouteRemovedCallback (called when a protocol deletes a route), and RoutingTableUpdatedCallback (called when the protocol updates the whole routing table at once). Protocols that use these callbacks will work with the *real routing* module. The *RealRouteUpdater* will register its route updating functions with each of the protocol's callbacks. Thus, every time one of the callbacks is called, the corresponding function in the *RealRouteUpdater* is also called and the received information is sent to the *RtNetlinkRouteUpdater* process which, in turn, updates the kernel routing tables.

The module also supports updating routes in a given network namespace, so it can also be used in the multiple nodes emulation architecture.

Finally, two other new features were added to the *EmuFdNetDeviceHelper* in the *fd-net-device* module to support the data plane in kernelspace architecture and multiple nodes emulation: setting a socket filter in the raw socket and selecting a network namespace for the socket.

With the *real routing* module implemented, a researcher must take the following steps to use the kernelspace architecture in a new protocol. When implementing the routing protocol in ns-3, the new callbacks must be called as appropriate. Then, when using the protocol in real environments, the simulation script needs the following changes: (1) set the socket filters in the *EmuFdNetDeviceHelper* and, if emulating multiple nodes, set the network namespace as well; (2) for each emulated routing node create an instance of the *RealRouteUpdater* and configure it with the routing protocol being used, the network namespace of the node, and the mapping between the ns-3 interfaces and the real interfaces of the host. With this configuration done, the created scenario will then be executed using the data plane in kernelspace architecture.

5.2.2 Performance Validation

The data plane in kernelspace architecture was validated using the Optimized Link State Routing Protocol (OLSR), a link state, proactive, L3 protocol [JC03].

The scalability in real environments was validated by comparing the performance of three different implementations: OLSR using traditional ns-3 emulation; OLSR using the real routing module; and olsrd, an implementation of OLSR for real systems [OLS].

The three scenarios have the same network topology (Figure 5.4), which is composed of three nodes connected by 100 Mbit/s Ethernet links:

- The source **S**. It runs OLSR and uses the `iperf` tool to generate UDP traffic destined to **D**. At the same time it also pings **D** to measure the round-trip time (RTT);
- The destination **D**. It runs OLSR and an `iperf` server to receive the UDP traffic from **S** and calculate the throughput and packet loss;
- An OLSR router **R**. It forwards the packets between **S** and **D**.

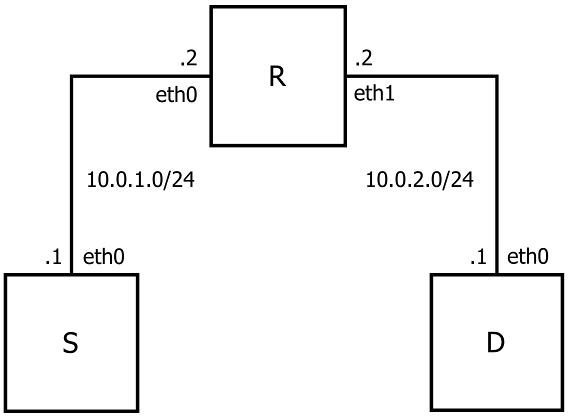

Figure 5.4: Network topology of the test scenarios

There were three scenarios tested. In the first (*ns-3 emulation*), **R** runs a traditional ns-3 emulation of OLSR. In the second scenario, (*ns-3 real routing*) **R** runs a ns-3 emulation of OLSR using the real routing module that was developed. In the last scenario (*olsrd*), **R** runs olsrd. In all scenarios, **S** and **D** run olsrd.

For each scenario a series of tests was executed, starting with **S** generating 1 Mbit/s of UDP traffic to **D** and then increasing that value to 2, 4, 8, 16, 32, 64, and 70.8 Mbit/s. Each test had a duration of 30 seconds and was repeated 5 times. The payload of the UDP packets was 160 bytes. The maximum offered data rate was of 70.8 Mbit/s because that is the maximum throughput that can be achieved in 100 Mbit/s Ethernet links for UDP packets with 160 bytes of data.

For each of these tests, four performance metrics were measured:

- The received data rate in **D**, which was given by the iperf tool;
- The packet loss ratio, which was also measured by iperf;
- The average round round-trip time, measured using the ping utility;
- The average CPU load in **R**, measured using the top utility.

The router **R** runs on an Intel Atom N270, which is a single core processor but that uses the Hyper-Threading technology so it behaves as having two logical cores. The operating system used was Ubuntu 14.04 LTS.

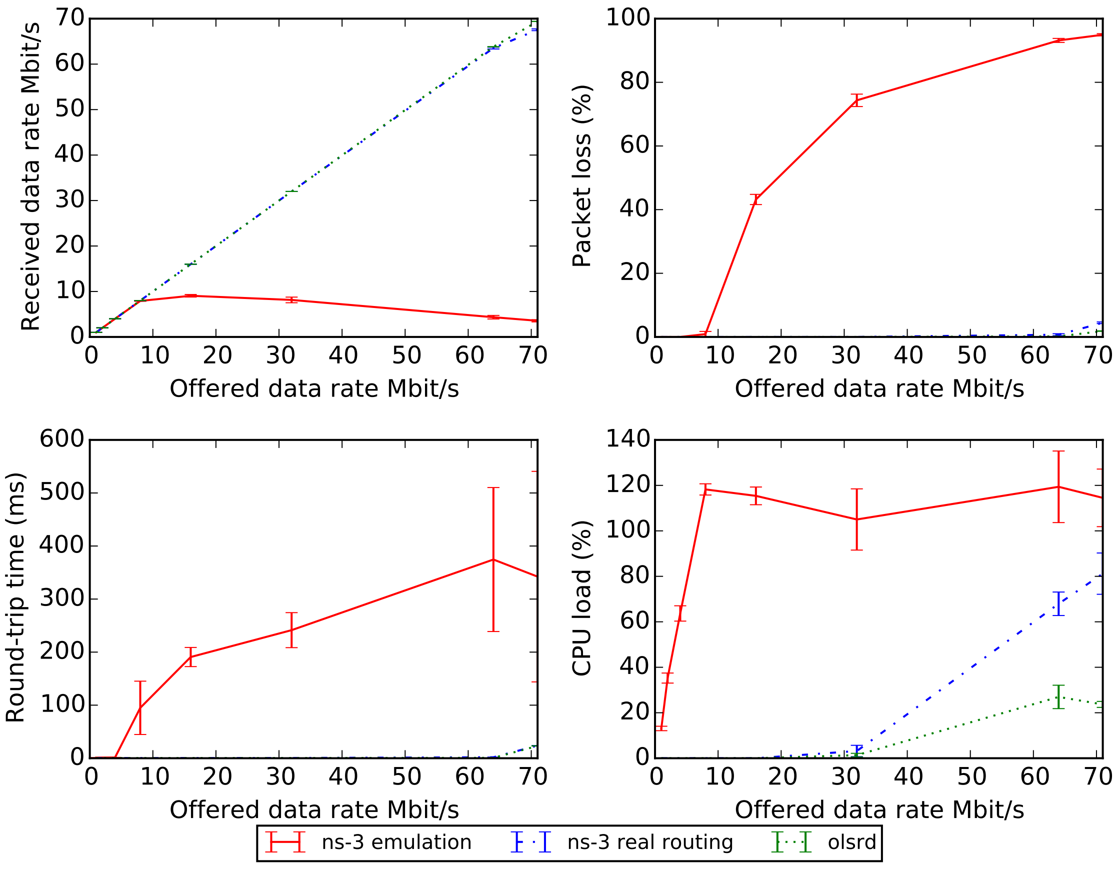

Figure 5.5: Data plane in kernelspace performance validation results for UDP traffic with payload of 160 bytes

The results are shown in Figure 5.5. The traditional ns-3 emulation implementation is able to forward packets at the offered rate until 8 Mbit/s. After that, performance starts degrading and when the offered rate hits 70.8 Mbit/s, ns-3 emulation has 95% packet loss and can only forward at 3.5Mbit/s. The round trip time (RTT) also increases to 342 ms, while until 8 Mbit/s it was around 1 ms. The reason why ns-3 emulation goes above 100% load is that the CPU of the router has two logical cores (due to Hyper-Threading) and ns-3 uses a dedicated thread for reading operations.

The real routing implementation and olsrd are able to forward packets at the offered rate at all rates except at 70.8 Mbit/s. At this highest rate there is packet loss of 4% and 2% for real routing and olsrd, respectively, and they forward traffic at 67.5 Mbit/s and 69 Mbit/s, respectively. Both of these implementations also provide very low round trip times, less than 0.5 ms until 32 Mbit/s, around 1 ms at 64 Mbit/s and around 23 ms at 70.8 Mbit/s. This means that, at the highest rate, real routing provides and increase in throughput of around 19 times when compared to traditional emulation as well as a decrease in RTT of around 14 times.

The reason that real routing performs slightly worse than olsrd is because it incurs a higher CPU load (around 3 times more at 70.8 Mbit/s). This can be attributed mainly to the socket filters, which must be executed for every packet that is received in the system. One way to reduce the overhead incurred by the filters is to enable them to be JIT (just-in-time) compiled, which means that instead of the kernel interpreting them for each packet, it will compile them to the underlying processor architecture just once and them run the compiled code for the received packets. However, we could not enable this optimization in the system we were using because it is a x86 machine and Ubuntu 14.04 only configures the kernel to allow JIT compilation of filters in x64 systems.

5.2.3 Code Reuse Validation

To validate that using the data plane in kernelspace architecture only requires writing a small amount of new code we counted the lines of code that were developed to use the real routing module with the existing ns-3 implementation of OLSR.

The results are presented in Figure 5.6, where *ns-3 emulation* indicates the number of lines of code of the existing ns-3 implementation of OLSR, and *ns-3 real routing* indicates the sum of those lines with the new lines that were developed to extend the implementation to work with the real routing module. The lines of code of the real routing module itself are not counted because it is a generic module that can be used with any proactive L3 protocol without having to be reimplemented. The blank and comment lines are not counted.

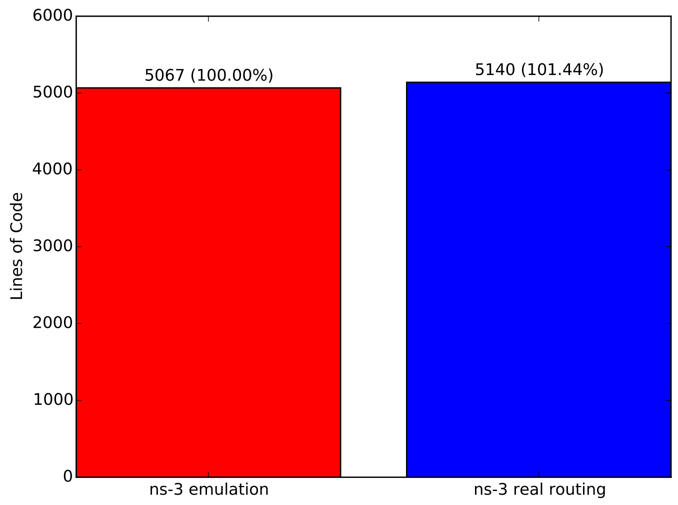

Figure 5.6: Code reuse validation results for data plane in kernelspace

It can be seen that only around 1.4% more lines of code were written to use the real routing module. Most of the new lines implement the code that calls the real routing module callbacks to update the routing table and the rest of the lines correspond to code that sets the socket filters and initializes the real routing module. We can thus conclude that the increase in code is marginal, especially when compared to the performance gains that are obtained when using the data plane in kernelspace architecture.

5.3 Multiple Nodes Emulation

5.3.1 Performance Validation

The data plane in kernelspace architecture for multiple nodes emulation was validated using the Optimized Link State Routing Protocol (OLSR).

The scalability in real environments was validated by comparing the performance of OLSR running on traditional ns-3 emulation of multiple nodes and OLSR using the real routing module and network namespaces.

The two scenarios have the same network topology (Figure 5.7), which is composed of three physical nodes connected by 100 Mbit/s Ethernet links:

- The source **S**. It runs OLSR and uses the `iperf` tool to generate UDP traffic destined to **D**. At the same time it also pings **D** to measure the round-trip time (RTT);
- The destination **D**. It runs OLSR and an `iperf` server to receive the UDP traffic from **S** and calculate the throughput and packet loss;
- A machine **R** which sits between **S** and **D**. It emulates a network of nodes that are connected in a line topology.

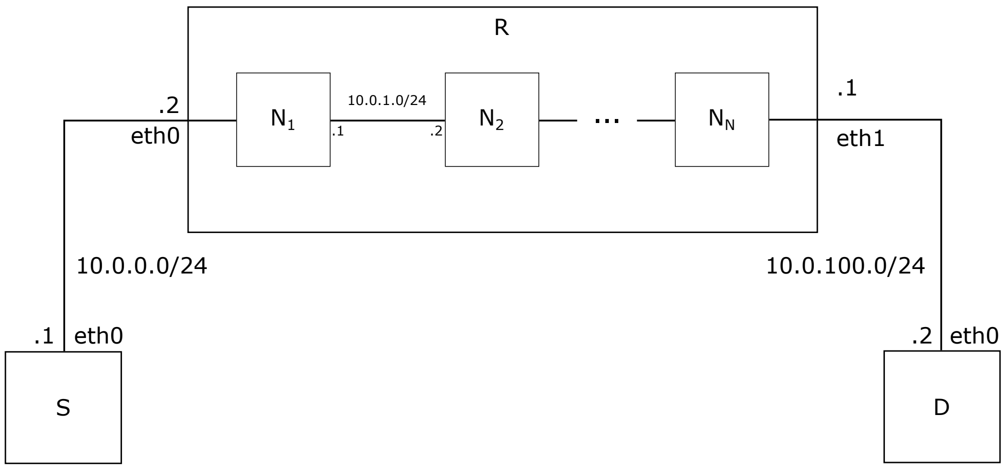

Figure 5.7: Network topology of the test scenarios

The line topology is used to determine how performance varies with the number of hops in a real scenario. There were two scenarios tested. In the first (*ns-3 emulation*), **R** runs a traditional ns-3 emulation of multiple OLSR routers. In the second scenario (*ns-3 real routing*), **R** runs a ns-3 emulation of multiple OLSR routers using the real routing module and the network namespaces architecture. In both scenarios, the links between the emulated nodes (simulated channels in ns-3 and virtual devices in network namespaces) were configured to have zero latency and 1 Gbit/s of maximum throughput.

For each scenario, two series of tests were executed to assess the UDP (first) and TCP (second) throughput. Both series started with **R** emulating a single node, and then increasing the number of emulated nodes in **R** to 2, 4, 8, 16, 24, and 32. Each test had a duration of 30 seconds and was repeated 5 times.

In the first series of tests **S** generates 16 Mbit/s of UDP traffic to **D**. This value was chosen because it is the maximum value that allows for some traffic to be received when using traditional emulation of 32 nodes. The payload of the UDP packets was 160 bytes.

For each of the tests in the first series, four performance metrics were measured:

- The received data rate in **D**, which was given by the `iperf` tool;
- The packet loss ratio, which was also measured by `iperf`;
- The average round round-trip time, measured using the `ping` utility;
- The average CPU load in **R**, measured using the `top` utility.

In the second series of tests, **S** generates TCP traffic to **D**. For each of the tests in this series, three performance metrics were measured:

- The received data rate in **D**, which was given by the `iperf` tool;
- The average round round-trip time, measured using the `ping` utility;
- The average CPU load in **R**, measured using the `top` utility.

The machine **R** runs on has an Intel Atom N270, which is a single core processor but that uses the Hyper-Threading technology so it behaves as having two logical cores. The operating system used was Ubuntu 14.04 LTS.

The results for UDP traffic are shown in Figure 5.8. Even with only one node, the traditional ns-3 emulation can only forward packets at 9.5 Mbit/s. With 8 emulated nodes there is already an 89% packet loss, a forwarding rate of less than 2 Mbit/s, and a RTT of more than 1.5 seconds. When the number of emulated nodes reach 32, the packet loss is 97% and the throughput is only 0.38 Mbit/s. RTT also peaks at 7.4 seconds.

The real routing implementation with network namespaces is able to achieve better performance. Until 8 emulated nodes it forwards traffic at the full 16 Mbit/s (8 times more than traditional emulation). The RTT is low as well, around 0.5 ms until 4 nodes and 2 ms for 8 nodes (750 times less than traditional emulation). With more than 8 nodes the performance starts decreasing and with 32 nodes, real routing only forwards at 1.5 Mbit/s (4 times more than traditional emulation) with a packet loss of 88%. RTT also increases to around 312 ms (24 times less than traditional emulation). The CPU load for real routing is lower up to 8 nodes. After that, both implementations are equivalent, but the real routing implementation is processing more traffic for the same load.

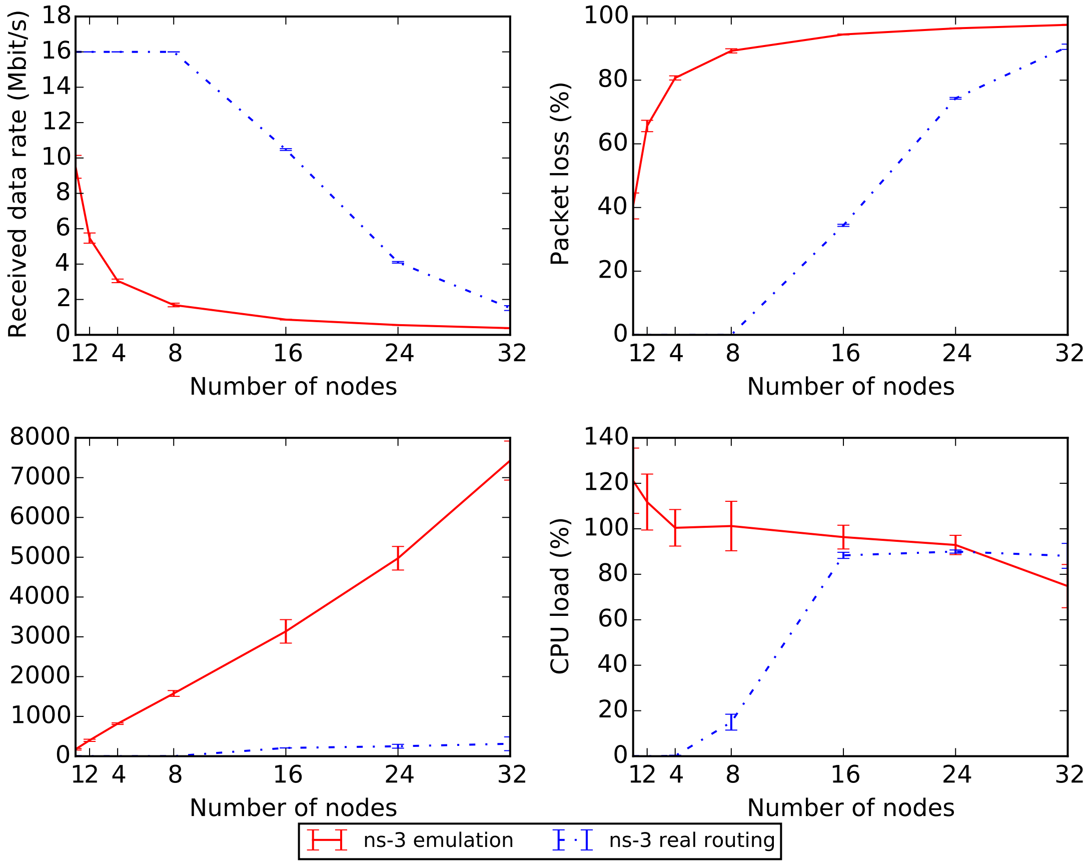

Figure 5.8: Multiple nodes emulation results for UDP traffic with payload of 160 bytes

The results for TCP traffic are shown in Figure 5.9. With only one node, traditional emulation can forward 55 Mbit/s of traffic. At 8 nodes, throughput decreases to 10 Mbit/s and RTT is 389 ms. With 32 nodes, traditional emulation can only achieve 2.6 Mbit/s and an RTT of 1.9 seconds.

Real routing can achieve throughputs of 94 Mbit/s with up to 8 nodes (9.4 times higher than traditional emulation) and 93 Mbit/s with 16 nodes. The RTT at 8 nodes is 38 ms (10 times lower than traditional emulation). When the number of nodes reaches 32, real routing can forward at a rate of 60 Mbit/s (23 times more than traditional emulation) and has a RTT of 124 ms (15 times less than traditional emulation). The CPU load of real routing is also lower until 8 nodes but is slightly higher with 16 or more nodes.

We can conclude that applying the data plane in kernelspace architecture with network namespaces to support multiple emulated nodes can provide increased performance for the same number of nodes of traditional emulation. Alternatively, the proposed architecture can also be used to allow increasing the number of nodes in an emulation while maintaining the same or better performance of traditional emulation.

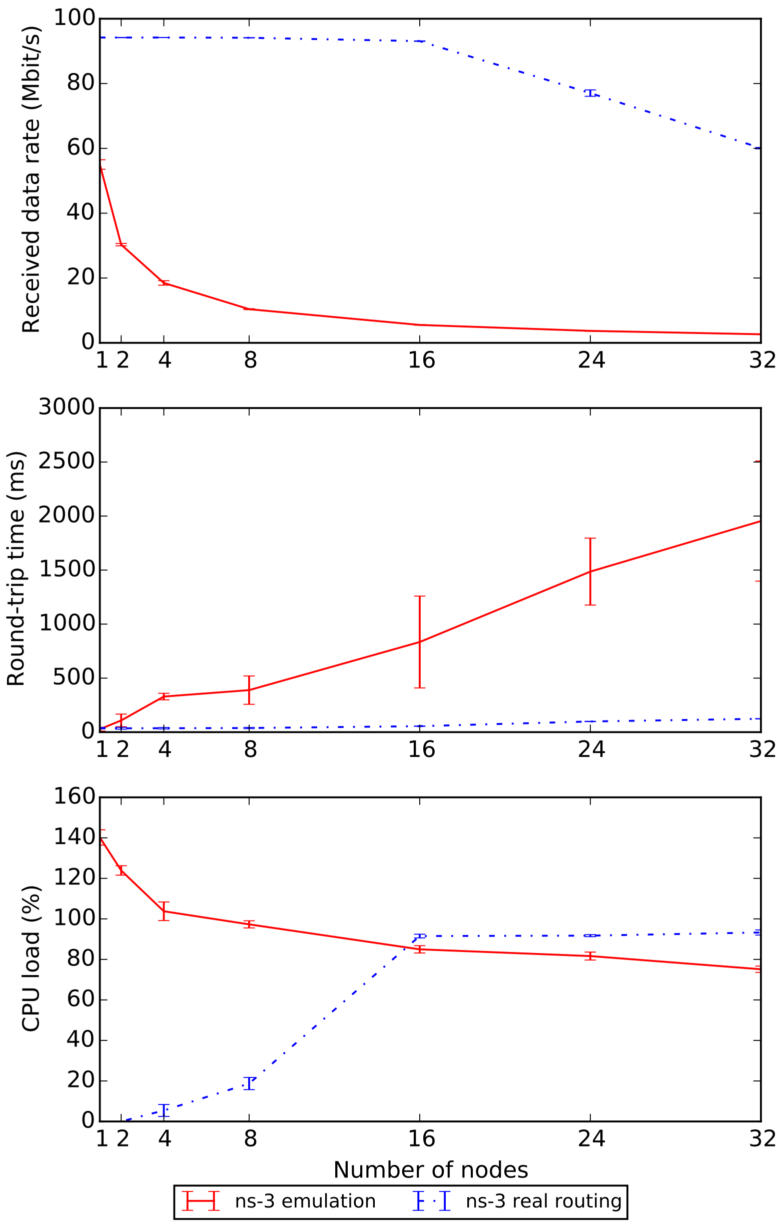

Figure 5.9: Multiple nodes emulation results for TCP traffic

Chapter 6

Conclusions and Future Work

The main goal of this dissertation is to propose an approach to shared protocol implementation that allows users of ns-3 to use their simulation code in real environments. We started by showing the problem that developers of new communication protocols face of having to port their protocol code between the simulator and a real system and then maintain both of those implementations during the protocol development process. To solve this problem we set the task of developing an architecture for shared protocol implementation that achieves the following goals: (1) full compatibility with ns-3, (2) scalability in simulation and in real environments, (3) large amount of code reuse between the simulator and the real system.

We then presented the current techniques and tools that provide a shared protocol implementation. These tools follow three main approaches: abstraction layers, direct code execution, and fast prototyping. Abstractions layers provide a development environment that hides from users the details of the underlying systems and can then run the developed code in each of those systems. Direct code execution provides an environment that allows code developed for real systems to be executed in simulations. This is done through the use of a virtualization layer. Finally, fast prototyping is a development model in which researchers implement the protocol directly in the simulator and then use emulation to allow the simulation code to be reused in testbeds. With this study we reached the conclusion that the best approach for our purposes is the use of fast prototyping with ns-3 emulation because it is the only solution that allows to use the full functionality of ns-3. It also has the advantage of providing high scalability in simulation since the protocols are developed directly in ns-3.

Despite ns-3 emulation being chosen as the best candidate to achieve the goals defined in this dissertation, it does not satisfy them all. In particular, it fails to achieve the goal of scalability in real environments. To better understand the limitations of ns-3 emulation performance we did an experiment that compares the forwarding performance between kernelspace, userspace, and ns-3 emulation. We showed that emulation in ns-3 introduces overhead in the processing of each packet and so its performance degrades quickly with the increase of the number of packets in the network.

The experiment also showed that userspace and kernelspace implementations suffer less from this problem.

Our proposed solution is thus to use fast prototyping with ns-3 emulation but move the data plane outside of the simulator. Since most of the traffic of typical networks is related to data, this solution improves the network nodes efficiency at the cost of having to port the data plane code. This is not a big problem, however, because the data plane is much simpler than the control plane resulting in a relatively small amount of code that needs to be ported. We proposed two architectures to implement this solution: executing the data plane in userspace and executing it in kernelspace. The former is more generic and works for any protocol while the latter is more efficient and allows more code reuse but is only applicable to proactive L3 protocols.

We also presented an architecture that enables the proposed optimizations to be used even when emulating multiple nodes in the same machine. This is done by using network namespaces, which are environments with the network resources isolated from each other.

In the case of the data plane in kernelspace architecture, since most of it can be reused between different protocols, we created a new ns-3 module called *real routing* which implements the generic parts of the architecture. This allows researchers to easily extend their ns-3 protocols to execute the data plane in kernelspace.

To validate the proposed solutions, we extended the WMRP and OLSR protocols to work with the developed architectures. Then, we compared the performance of the new architectures against traditional ns-3 emulation, both for single and multiple emulated nodes in the same physical node, running multiple tests with different configurations. The summarized interpretation of the obtained results is the following: emulating a single ns-3 node, the maximum throughput can be improved by as much as 4.9 times in userspace and 19 times in kernelspace, while having the RTT lowered by 5.3 and 14 times, respectively; when emulating multiple ns-3 nodes, the maximum throughput can be improved, in kernelspace, by as much as 23 times while having 15 times lower RTT. With this results in consideration, we can conclude that the proposed architectures (userspace and kernelspace) allow much better data plane performance, when compared against traditional ns-3 emulation. The kernelspace architecture has the best performance, obtaining results very close to a real protocol implementation. Thus, when possible, the kernelspace architecture should be used to obtain the best emulation performance results. This enables the use of fast prototyping for network protocols in more traffic demanding scenarios or in real nodes with limited processing power.

The amount of code reuse obtained was also verified by counting the lines of code needed for each implementation. We found that implementing the data plane in userspace only required the development of around 11% additional lines of code on top of traditional emulation implementations. In the case of executing the data plane in kernelspace, that value can be reduce to 1.4% by using the *real routing* module.

In conclusion, the objectives of this work were fully achieved, complemented by validation results that demonstrated both the relevance of the problem addressed by the work and the quality of the proposed solution. As a result of this work, ns-3 users will be able to reuse their simulation code in real networks, now with contained packet processing overhead, enabling them to reap the benefits of the fast prototyping process without the previously associated negative performance impact.

6.1 Main Contributions

The main contributions of our work are the following:

1. The proposal of two architectures that execute the data plane of routing protocols outside of the simulator in order to improve the performance of emulated nodes. One architecture executes the data plane in a userspace process while the other executes the data plane in kernelspace by updating the routing tables of the kernel;

2. An architecture that uses network namespaces to enable the use of the previous two architectures in scenarios where multiple nodes need to be emulated in a single machine;

3. A new ns-3 module, *real routing*, which allows researchers to easily extend their routing protocols to use the data plane in kernelspace architecture;

4. Empirical studies that validate the proposed architectures' scalability in real environments and the amount of code reuse that can be obtained.

6.2 Future Work

The main objectives of this dissertation were achieved, but this work revealed some topics that are relevant and should be addressed as future work, answering some questions raised in this thesis and improving the obtained results.

First, the *real routing* module can be proposed for integration in the main ns-3 distribution in order to reach more protocol developers and researchers. It can also be made more easy to use, for example, it could allow socket filters to be specified as a string with the tcpdump filter syntax instead of an array of machine code that must be manually generated by users. Another helpful feature would be to automatically generate the network namespaces and their connections based on the topology of the nodes created in the simulator. At the moment, users have to manually create the namespaces and configure the emulated nodes with their corresponding namespace.

To increase the usefulness of the *real routing* module, it can also be extended to support reactive protocols. To do this, a kernel module or other facility would have to be created that would listen to the events generated in the kernel during packet processing (e.g. failure to find a route) and send the relevant information back to the simulator so the appropriate control plane operations can be executed.

It would be also interesting to test the performance gains that can be obtained with the data plane in kernelspace architecture on machines that support JIT compilation of socket filters.

Finally, regarding the architecture for multiple nodes emulation, the performance and reliability of the netem kernel module should be validated to determine if it can be used to emulate in the virtual devices the network conditions that are defined in ns-3. Additionally, other types of virtualization environments (e.g. OpenVZ or LXC) could be tested to see whether they can obtain better performance than network namespaces combined with `veth` virtual network devices.

References

[AR12] John Abraham and George Riley. Simulator-agnostic ns-3 applications. In *Proceedings of the 5th International ICST Conference on Simulation Tools and Techniques*, SIMUTOOLS '12, pages 391–396, Sirmione-Desenzano, Italy, 2012. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[CFR11] Gustavo Carneiro, Helder Fontes, and Manuel Ricardo. Fast prototyping of network protocols through ns-3 simulation model reuse. *Simulation Modelling Practice and Theory*, 19(9):2063–2075, October 2011.

[Edg14] Jake Edge. Namespaces in operation, part 7: Network namespaces. https://lwn.net/Articles/580893, January 2014. Accessed June 01, 2015.

[Ins01] Gianluca Insolvibile. Kernel korner: Linux socket filter: Sniffing bytes over the network. *Linux J.*, 2001(86):8–, June 2001.

[Jan08] Samuel Thomas Jansen. *Network Simulation Cradle*. Thesis, The University of Waikato, 2008.

[JC03] P. Jacquet and T. Clausen. Optimized link state routing protocol (OLSR). RFC 3626, IETF, October 2003.

[KBO12] A.R. Khan, S.M. Bilal, and M. Othman. A performance comparison of open source network simulators for wireless networks. In *2012 IEEE International Conference on Control System, Computing and Engineering (ICCSCE)*, pages 34–38, Penang, Malaysia, November 2012.

[KLW10] Georg Kunz, Olaf Landsiedel, and Georg Wittenburg. From Simulations to Deployments. In Klaus Wehrle, Mesut Günes, and James Gross, editors, *Modeling and Tools for Network Simulation*, pages 83–97. Springer, New York, 2010 edition edition, June 2010.

[KMC$^+$00] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click Modular Router. *ACM Trans. Comput. Syst.*, 18(3):263–297, August 2000.

[Lac10] Mathieu Lacage. *Experimentation Tools for Networking Research*. Ph.D., Universite de Nice-Sophia Antipolis, 2010.

[MLL$^+$09] Shivkumar C. Muthukumar, Xiaozhou Li, Changbin Liu, Joseph B. Kopena, Mihai Oprea, and Boon Thau Loo. Declarative toolkit for rapid network protocol simulation and experimentation. In *ACM SIGCOMM Conference on Data Communications (demo)*, Barcelona, Spain, August 2009.

[Mor10] J. Paul Morrison. *Flow-Based Programming, 2nd Edition: A New Approach to Application Development*. CreateSpace Independent Publishing Platform, Unionville, Ont., 2 edition edition, May 2010.

[Moy98] J. Moy. OSPF version 2. RFC 2328, IETF, April 1998.

[NJG02] Michael Neufeld, Ashish Jain, and Dirk Grunwald. Nsclick:: Bridging Network Simulation and Deployment. In *Proceedings of the 5th ACM International Workshop on Modeling Analysis and Simulation of Wireless and Mobile Systems*, MSWiM '02, pages 74–81, New York, NY, USA, 2002. ACM.

[ns-a] ns-3 home page. http://www.nsnam.org/. Accessed January 16, 2015.

[ns-b] ns-3.21 data collection manual. http://www.nsnam.org/docs/release/3.21/manual/html/data-collection.html. Accessed January 27, 2015.

[ns-c] ns-3.21 emulation overview. http://www.nsnam.org/docs/release/3.21/models/html/emulation-overview.html. Accessed January 16, 2015.

[ns-d] ns-3.21 tracing manual. http://www.nsnam.org/docs/release/3.21/manual/html/tracing.html. Accessed January 27, 2015.

[OLS] OLSR.org. OLSRd. http://www.olsr.org/. Accessed June 01, 2015.

[PBRD03] C. Perkins, E. Belding-Royer, and S. Das. Ad hoc on-demand distance vector (AODV) routing. RFC 3561, IETF, July 2003.

[RCF+10] M. Ricardo, G. Carneiro, P. Fortuna, F. Abrantes, and J. Dias. WiMetroNet a scalable wireless network for metropolitan transports. In *2010 Sixth Advanced International Conference on Telecommunications (AICT)*, pages 520–525, Barcelona, Spain, May 2010.

[SKKK03] J. Salim, H. Khosravi, A. Kleen, and A. Kuznetsov. Linux netlink as an IP services protocol. RFC 3549, IETF, July 2003.

[SM11] P Lalith Suresh and Ruben Merz. ns-3-click: click modular router integration for ns-3. In *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*, pages 423–430, Barcelona, Spain, 2011. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[The09] The Linux Foundation. netem. http://www.linuxfoundation.org/collaborate/workgroups/networking/netem, November 2009. Accessed June 01, 2015.

[TUM+13] Hajime Tazaki, Frédéric Uarbani, Emilio Mancini, Mathieu Lacage, Daniel Camara, Thierry Turletti, and Walid Dabbous. Direct Code Execution: Revisiting Library OS Architecture for Reproducible Network Experiments. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '13, pages 217–228, New York, NY, USA, 2013. ACM.

[TUT13] Hajime Tazaki, Frédéric Urbani, and Thierry Turletti. DCE cradle: Simulate network protocols with real stacks for better realism. In *Proceedings of the 6th International ICST Conference on Simulation Tools and Techniques*, SimuTools '13, pages 153–158, Cannes, France, 2013. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[U.S] U.S. NRL Networks and Communications Systems Branch. The Protean Protocol Prototyping Library (Protolib). http://www.nrl.navy.mil/itd/ncs/products/protolib. Accessed January 16, 2015.

[WvLW09] E. Weingartner, H. vom Lehn, and K. Wehrle. A performance comparison of recent network simulators. In *IEEE International Conference on Communications, 2009. ICC '09*, pages 1–5, Dresden, Germany, June 2009.

REFERENCES

Appendix A

Detailed Results of Performance Evaluations

This chapter provides the detailed results in table form of all the evaluations that were presented. All confidence intervals shown have a 95% confidence level.

A.1 Results of Evaluation of ns-3 Emulation Perfomance

Table A.1: Performance results of ns-3 emulation forwarding of UDP packets with payload of 1470 bytes

Offered Rate (Mbit/s)	**Received Rate (Mbit/s)**	**Packet Loss (%)**	**RTT (ms)**	**CPU Load (%)**
1	1.00 ± 0.00	0.00 ± 0.00	3.00 ± 0.00	3.00 ± 0.00
2	2.00 ± 0.00	0.00 ± 0.00	5.00 ± 0.00	5.00 ± 0.00
4	4.00 ± 0.00	0.00 ± 0.00	9.60 ± 0.68	9.60 ± 0.68
8	8.00 ± 0.00	0.00 ± 0.00	18.60 ± 0.68	18.60 ± 0.68
16	16.00 ± 0.00	0.00 ± 0.00	35.00 ± 0.88	35.00 ± 0.88
32	32.04 ± 0.00	0.00 ± 0.00	62.40 ± 0.68	62.40 ± 0.68
64	64.26 ± 0.01	0.00 ± 0.01	97.40 ± 2.08	97.40 ± 2.08
90	90.34 ± 0.05	0.13 ± 0.06	122.20 ± 4.06	122.20 ± 4.06

Table A.2: Performance results of userspace forwarding of UDP packets with payload of 1470 bytes

Offered Rate (Mbit/s)	**Received Rate (Mbit/s)**	**Packet Loss (%)**	**RTT (ms)**	**CPU Load (%)**
1	1.00 ± 0.00	0.00 ± 0.00	0.53 ± 0.02	0.00 ± 0.00
2	2.00 ± 0.00	0.00 ± 0.00	0.53 ± 0.02	1.00 ± 0.00
4	4.00 ± 0.00	0.00 ± 0.00	0.55 ± 0.05	3.00 ± 0.00
8	8.00 ± 0.00	0.00 ± 0.00	0.56 ± 0.01	5.40 ± 0.68
16	16.00 ± 0.00	0.00 ± 0.00	0.60 ± 0.02	11.00 ± 0.00
32	32.04 ± 0.00	0.00 ± 0.00	0.67 ± 0.02	22.00 ± 0.00
64	64.26 ± 0.00	0.00 ± 0.00	0.78 ± 0.06	39.00 ± 1.24
90	90.48 ± 0.03	0.00 ± 0.00	1.05 ± 0.24	45.00 ± 1.52

Table A.3: Performance results of kernelspace forwarding of UDP packets with payload of 1470 bytes

Offered Rate (Mbit/s)	**Received Rate (Mbit/s)**	**Packet Loss (%)**	**RTT (ms)**	**CPU Load (%)**
1	1.00 ± 0.00	0.00 ± 0.00	0.29 ± 0.01	0.00 ± 0.00
2	2.00 ± 0.00	0.00 ± 0.00	0.29 ± 0.01	0.00 ± 0.00
4	4.00 ± 0.00	0.00 ± 0.00	0.30 ± 0.00	0.00 ± 0.00
8	8.00 ± 0.00	0.00 ± 0.00	0.32 ± 0.02	0.00 ± 0.00
16	16.00 ± 0.00	0.00 ± 0.00	0.36 ± 0.01	0.00 ± 0.00
32	32.04 ± 0.00	0.00 ± 0.00	0.43 ± 0.01	0.00 ± 0.00
64	64.26 ± 0.00	0.00 ± 0.00	0.52 ± 0.01	0.00 ± 0.00
90	90.46 ± 0.00	0.00 ± 0.00	0.55 ± 0.03	0.00 ± 0.00

Table A.4: Performance results of ns-3 emulation forwarding of UDP packets with payload of 160 bytes

Offered Rate (Mbit/s)	**Received Rate (Mbit/s)**	**Packet Loss (%)**	**RTT (ms)**	**CPU Load (%)**
1	1.00 ± 0.00	0.00 ± 0.00	18.60 ± 0.68	18.60 ± 0.68
2	2.00 ± 0.00	0.00 ± 0.00	36.20 ± 1.04	36.20 ± 1.04
4	4.00 ± 0.00	0.00 ± 0.00	61.60 ± 1.67	61.60 ± 1.67
8	8.00 ± 0.00	0.00 ± 0.00	97.00 ± 4.39	97.00 ± 4.39
16	14.38 ± 0.93	10.09 ± 5.77	138.20 ± 7.42	138.20 ± 7.42
32	12.74 ± 1.10	59.95 ± 3.58	125.40 ± 12.44	125.40 ± 12.44
64	7.99 ± 0.19	87.39 ± 0.30	112.80 ± 6.36	112.80 ± 6.36
71	6.79 ± 0.16	90.37 ± 0.28	109.60 ± 4.35	109.60 ± 4.35

Table A.5: Performance results of userspace forwarding of UDP packets with payload of 160 bytes

Offered Rate (Mbit/s)	**Received Rate (Mbit/s)**	**Packet Loss (%)**	**RTT (ms)**	**CPU Load (%)**
1	1.00 ± 0.00	0.00 ± 0.00	0.45 ± 0.03	5.80 ± 0.56
2	2.00 ± 0.00	0.00 ± 0.00	0.45 ± 0.04	11.00 ± 0.00
4	4.00 ± 0.00	0.00 ± 0.00	0.53 ± 0.10	22.00 ± 0.00
8	8.00 ± 0.00	0.00 ± 0.00	0.46 ± 0.04	33.20 ± 0.56
16	15.99 ± 0.02	0.06 ± 0.12	0.61 ± 0.20	53.40 ± 0.68
32	25.46 ± 0.24	20.15 ± 0.61	31.34 ± 24.88	93.20 ± 3.33
64	31.67 ± 3.38	50.15 ± 5.28	16.69 ± 0.75	91.00 ± 2.32
71	27.76 ± 5.61	60.56 ± 8.12	36.63 ± 51.16	88.20 ± 10.80

Table A.6: Performance results of kernelspace forwarding of UDP packets with payload of 160 bytes

Offered Rate (Mbit/s)	**Received Rate (Mbit/s)**	**Packet Loss (%)**	**RTT (ms)**	**CPU Load (%)**
1	1.00 ± 0.00	0.00 ± 0.00	0.29 ± 0.01	0.00 ± 0.00
2	2.00 ± 0.00	0.00 ± 0.00	0.29 ± 0.01	0.00 ± 0.00
4	4.00 ± 0.00	0.00 ± 0.00	0.29 ± 0.01	0.00 ± 0.00
8	8.00 ± 0.00	0.00 ± 0.00	0.29 ± 0.01	0.00 ± 0.00
16	16.00 ± 0.00	0.00 ± 0.00	0.30 ± 0.01	0.00 ± 0.00
32	31.99 ± 0.01	0.03 ± 0.03	0.32 ± 0.02	0.00 ± 0.00
64	63.09 ± 0.27	1.15 ± 0.14	0.51 ± 0.04	5.00 ± 3.66
71	68.45 ± 0.50	3.18 ± 0.64	23.03 ± 0.52	7.96 ± 1.31

A.2 Results of Evaluation of Data Plane in Userspace Architecture

Table A.7: Performance results of traditional emulation of WMRP

Offered Rate (Mbit/s)	Received Rate (Mbit/s)	Packet Loss (%)	RTT (ms)	CPU Load (%)
1	1.00 ± 0.00	0.00 ± 0.00	1.41 ± 0.03	0.09 ± 0.23
2	2.00 ± 0.00	0.00 ± 0.00	1.41 ± 0.06	11.10 ± 6.70
4	4.00 ± 0.00	0.00 ± 0.00	1.50 ± 0.06	41.60 ± 9.57
8	8.00 ± 0.00	0.00 ± 0.00	2.02 ± 0.63	84.21 ± 7.03
16	14.43 ± 0.64	9.77 ± 4.09	65.35 ± 18.87	125.63 ± 10.80
32	11.80 ± 0.11	62.88 ± 0.42	110.08 ± 8.70	104.22 ± 4.13
64	10.49 ± 1.12	83.45 ± 1.77	101.28 ± 37.76	112.14 ± 26.06
71	9.27 ± 0.09	86.83 ± 0.08	84.60 ± 22.36	104.60 ± 5.17

Table A.8: Performance results of data plane in userspace with WMRP

Offered Rate (Mbit/s)	Received Rate (Mbit/s)	Packet Loss (%)	RTT (ms)	CPU Load (%)
1	1.00 ± 0.00	0.00 ± 0.00	0.91 ± 0.02	0.00 ± 0.00
2	2.00 ± 0.00	0.00 ± 0.00	0.89 ± 0.02	0.00 ± 0.00
4	4.00 ± 0.00	0.00 ± 0.00	0.90 ± 0.06	3.73 ± 5.80
8	8.00 ± 0.00	0.00 ± 0.00	0.96 ± 0.11	22.44 ± 17.46
16	15.99 ± 0.01	0.04 ± 0.04	1.06 ± 0.13	13.43 ± 13.55
32	31.62 ± 0.02	1.19 ± 0.06	2.07 ± 0.21	71.01 ± 5.51
64	43.44 ± 1.90	31.84 ± 3.08	7.90 ± 0.47	98.00 ± 15.91
71	43.84 ± 4.79	37.77 ± 6.74	16.25 ± 1.02	99.30 ± 7.81

A.3 Results of Evaluation of Data Plane in Kernelspace Architecture

Table A.9: Performance results of traditional emulation of OLSR

Offered Rate (Mbit/s)	**Received Rate (Mbit/s)**	**Packet Loss (%)**	**RTT (ms)**	**CPU Load (%)**
1	1.00 ± 0.00	0.00 ± 0.00	1.06 ± 0.07	13.09 ± 0.97
2	2.00 ± 0.00	0.00 ± 0.00	1.20 ± 0.10	35.26 ± 2.21
4	4.00 ± 0.00	0.00 ± 0.00	1.38 ± 0.16	63.68 ± 3.37
8	7.92 ± 0.07	0.89 ± 0.84	94.97 ± 50.26	118.23 ± 2.47
16	9.06 ± 0.28	43.19 ± 1.60	190.75 ± 18.03	115.39 ± 3.89
32	8.14 ± 0.62	74.34 ± 1.95	241.43 ± 33.00	105.01 ± 13.47
64	4.34 ± 0.39	93.16 ± 0.63	374.54 ± 135.71	119.38 ± 15.74
71	3.55 ± 0.24	94.94 ± 0.35	342.29 ± 198.42	114.52 ± 12.67

Table A.10: Performance results of data plane in kernelspace with OLSR

Offered Rate (Mbit/s)	**Received Rate (Mbit/s)**	**Packet Loss (%)**	**RTT (ms)**	**CPU Load (%)**
1	1.00 ± 0.00	0.00 ± 0.00	0.37 ± 0.02	0.00 ± 0.00
2	2.00 ± 0.00	0.00 ± 0.00	0.37 ± 0.01	0.00 ± 0.00
4	4.00 ± 0.00	0.00 ± 0.00	0.37 ± 0.00	0.00 ± 0.00
8	8.00 ± 0.00	0.00 ± 0.00	0.37 ± 0.01	0.00 ± 0.00
16	16.00 ± 0.00	0.00 ± 0.00	0.39 ± 0.01	0.00 ± 0.00
32	31.99 ± 0.01	0.00 ± 0.01	0.45 ± 0.04	3.17 ± 2.50
64	63.53 ± 0.20	0.71 ± 0.31	1.82 ± 0.26	67.93 ± 5.17
71	67.56 ± 0.19	4.46 ± 0.26	23.80 ± 0.10	81.18 ± 9.08

Table A.11: Performance results of olsrd

Offered Rate (Mbit/s)	**Received Rate (Mbit/s)**	**Packet Loss (%)**	**RTT (ms)**	**CPU Load (%)**
1	1.00 ± 0.00	0.00 ± 0.00	0.36 ± 0.01	0.00 ± 0.00
2	2.00 ± 0.00	0.00 ± 0.00	0.38 ± 0.01	0.00 ± 0.00
4	4.00 ± 0.00	0.00 ± 0.00	0.37 ± 0.01	0.00 ± 0.00
8	8.00 ± 0.00	0.00 ± 0.00	0.37 ± 0.01	0.00 ± 0.00
16	16.00 ± 0.00	0.00 ± 0.00	0.37 ± 0.01	0.00 ± 0.00
32	32.00 ± 0.00	0.00 ± 0.01	0.40 ± 0.01	1.35 ± 0.80
64	63.82 ± 0.02	0.18 ± 0.03	0.68 ± 0.06	26.96 ± 5.14
71	69.39 ± 0.02	1.84 ± 0.07	23.16 ± 0.29	23.67 ± 1.32

A.4 Results of Evaluation of Multiple Nodes Emulation

Table A.12: Performance results of multiple nodes traditional emulation for UDP traffic at 16 Mbit/s

Number of Nodes	Received Rate (Mbit/s)	Packet Loss (%)	RTT (ms)	CPU Load (%)
1	9.50 ± 0.65	40.49 ± 4.07	175.04 ± 19.85	121.14 ± 14.36
2	5.47 ± 0.29	65.61 ± 1.79	399.11 ± 27.60	111.77 ± 12.30
4	3.05 ± 0.10	80.70 ± 0.65	817.62 ± 19.83	100.44 ± 8.03
8	1.69 ± 0.10	89.21 ± 0.66	1576.68 ± 72.79	101.22 ± 10.87
16	0.86 ± 0.02	94.36 ± 0.13	3136.98 ± 295.01	96.35 ± 5.22
24	0.55 ± 0.01	96.28 ± 0.10	4974.37 ± 296.08	92.89 ± 4.24
32	0.38 ± 0.02	97.38 ± 0.13	7428.99 ± 490.58	74.80 ± 9.51

Table A.13: Performance results of multiple nodes with data plane in kernelspace for UDP traffic at 16 Mbit/s

Number of Nodes	Received Rate (Mbit/s)	Packet Loss (%)	RTT (ms)	CPU Load (%)
1	16.00 ± 0.00	0.00 ± 0.00	0.38 ± 0.03	0.03 ± 0.09
2	16.00 ± 0.00	0.00 ± 0.00	0.42 ± 0.01	0.00 ± 0.00
4	16.00 ± 0.00	0.00 ± 0.00	0.54 ± 0.04	0.15 ± 0.27
8	16.00 ± 0.00	0.00 ± 0.00	2.29 ± 0.54	14.97 ± 3.49
16	10.48 ± 0.05	34.40 ± 0.33	207.89 ± 1.58	88.32 ± 1.35
24	4.10 ± 0.04	74.30 ± 0.28	250.02 ± 48.44	90.00 ± 0.69
32	1.51 ± 0.14	90.49 ± 0.83	312.41 ± 173.29	88.10 ± 5.48

Table A.14: Performance results of multiple nodes traditional emulation for TCP traffic

Number of Nodes	**Received Rate (Mbit/s)**	**RTT (ms)**	**CPU Load (%)**
1	55.02 ± 1.47	27.58 ± 17.31	140.20 ± 3.78
2	30.31 ± 0.37	108.49 ± 58.50	123.91 ± 2.33
4	18.49 ± 0.69	329.44 ± 30.26	103.76 ± 4.59
8	10.39 ± 0.14	388.97 ± 131.37	97.30 ± 1.77
16	5.52 ± 0.08	834.28 ± 424.88	84.96 ± 1.81
24	3.69 ± 0.10	1485.97 ± 309.77	81.64 ± 1.96
32	2.64 ± 0.06	1953.64 ± 556.10	75.14 ± 1.55

Table A.15: Performance results of multiple nodes with data plane in kernelspace for TCP traffic

Number of Nodes	**Received Rate (Mbit/s)**	**RTT (ms)**	**CPU Load (%)**
1	94.20 ± 0.02	37.16 ± 5.80	0.00 ± 0.00
2	94.19 ± 0.04	35.69 ± 8.37	0.00 ± 0.00
4	94.20 ± 0.01	36.36 ± 6.51	5.43 ± 2.95
8	94.12 ± 0.04	38.12 ± 5.50	18.72 ± 3.01
16	93.09 ± 0.06	55.40 ± 1.20	91.52 ± 0.92
24	77.05 ± 0.99	98.78 ± 1.54	91.73 ± 0.49
32	60.07 ± 0.22	123.84 ± 0.97	93.28 ± 1.28